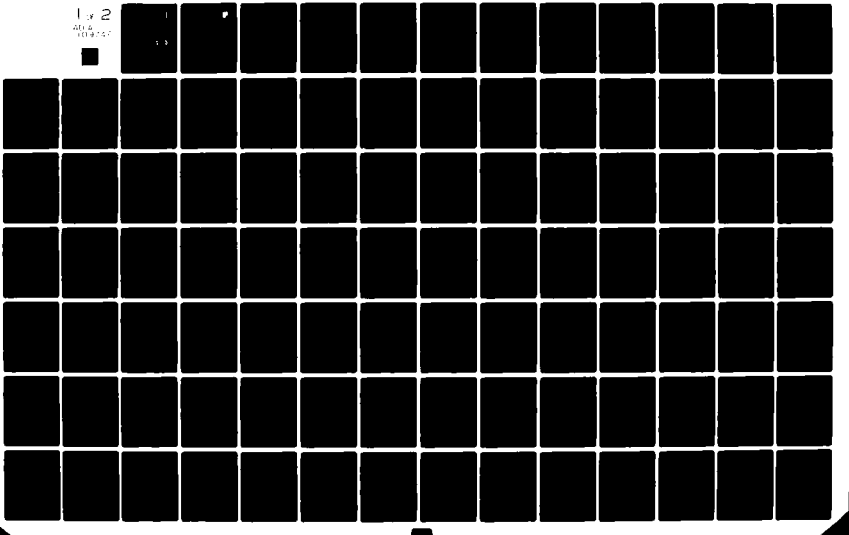
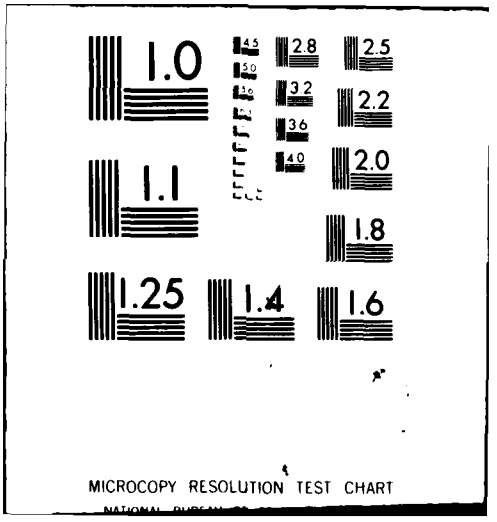
AD-A109 747 COMPUTER SCIENCES CORP FALLS CHURCH VA F/G 9/2
ADA INTEGRATED ENVIRONMENT II.(U)
DEC 81 F30602-80-C-0292
UNCLASSIFIED RADC-TR-81-363 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

PHOTOGRAPH THIS SHEET

AD A109747

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Computer Sciences Corp
Falls Church, VA

ADA Integrated Environment II

DOCUMENT IDENTIFICATION

Dec. 81

Contract F30602-80-C-0292

RADC-TR-81-363

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS	GRA&I
DTIC	TAB
UNANNOUNCED	
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
ELECTE
S JAN 19 1982 D
D

DATE ACCESSIONED

82 01 12 004

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-363
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT II

Computer Sciences Corporation

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-363 has been reviewed and is approved for publication.

APPROVED: Donald F. Roberts

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC. (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-363	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT II		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0292
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Corporation 803 Broad Street Falls Church VA 22046		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/62702F/33126F 55811918
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981 13. NUMBER OF PAGES 174
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This report describes the rationale of the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an APSE is

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this report include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

INTRODUCTION

This document describes the Computer Sciences Corporation (CSC)/Software Engineering Associates (SEA) team's approach to the design of a Minimal Ada Programming Support Environment (MAPSE) under the Ada Integrated Environment (AIE) contract with Rome Air Development Center (RADC). Design issues that influenced the preliminary design are delineated and the system design is presented with its rationale. System requirements are included in the System Specification (Type A) while the details of the preliminary design are given in the Computer Program Development Specifications (Type B5), both of which accompany this document.

This Interim Technical Report addresses the overall system design as well as the design of the individual components. The first section is an introduction, containing an executive summary of the AIE project, a discussion of the principles that guided our system design, an overview of the system requirements, and a description of the system design in terms of functionality and system interfaces. Subsequent sections of this document address each major component of the system individually and provide an introduction to the component, the design principles followed during this preliminary design phase, the background for the chosen design, and a functional description of the component, including definition of system interfaces and rationale for the design decisions. Each of these parts is intended to be read and understood independently of the others.

SECTION 1 - EXECUTIVE SUMMARY

While the need for a common higher order programming language (HOL) has been recognized over the past several years, the need for automated management and development support tools has also become apparent. From development and maintenance standpoints, facilities that enhance the use of a common HOL reduce costs. If, in addition, the language were retargetable and the environment tools that support the language development were rehostable, then use of the language and its support system would be enhanced. To this end, a preliminary requirements definition, PEBBLEMAN, was developed in 1978 and revised in January 1979. Subsequently, the STONEMAN document (issued in November 1979 and revised in February 1980) described the basic design goals for an Ada Program Support Environment (APSE). STONEMAN, along with the SOW issued by RADC under contract number F30602-80-C-0292, and the Reference Manual for the Ada Programming Language (July 1980), form the basis for this effort.

The CSC/SEA design for the AIE provides the first MAPSE in accordance with the STONEMAN precepts by providing project managers and programmers with a set of modern techniques and tools to support software development. The requirement for a totally integrated, highly portable, language-oriented development system as defined by this effort will result in the implementation of a sorely needed, readily achievable, advancement to the state-of-the-art of software development environments. The Ada compiler and its associated language development and project management tools, coupled with the design of a machine-independent virtual operating system to provide required portability, will result in the implementation for the first time of a truly integrated approach to solving the major software development problems that confront both industry and the Department of Defense (DoD).

The MAPSE design is similiar in concept to the highly-acclaimed UNIX™ timesharing system, which provides a fully-integrated systems approach to providing a software development environment for the C language. The MAPSE will do the same for Ada. It must be recognized, however, that the far-reaching requirements for the Ada environment entail considerable extensions to the UNIX concept. The requirements for a unified set of Ada

language tools, for a sophisticated attributed data base, for project and configuration management facilities, and for Ada tasking support all contribute to the need for an integrated system design. The additional requirement for a fully rehostable system mandates the development of a comprehensive virtual operating system to supply an Ada integrated environment with uniform and standard interfaces.

The MAPSE incorporates a specific set of Ada tools to support software development. This minimal set of common tools is extensible so that the MAPSE can expand and change as additional needs and uses are identified. The MAPSE is designed to be portable: user interfaces are machine-independent and the MAPSE is designed to present identical user interfaces on differing host machines. All machine-dependent features, such as low-level input/output (I/O), are isolated. The MAPSE to be developed under this contract includes the Kernel APSE (KAPSE) and a Tool Set. The KAPSE is composed of a machine-independent user interface, run-time support facilities, communications facilities, a data base, and a machine dependent interface. The Tool Set consists of a Text Editor, Ada Compiler, Symbolic Debugger, Linker, Command Language Interpreter, and Configuration Manager. This design of the MAPSE accommodates extension to a full integrated support environment by the addition of tools through a common interface.

This Interim Technical Report (ITR) highlights the principles emphasized in the design of the CSC/SEA MAPSE. All STONEMAN design goals are achieved in this design, and the system is fully compliant with the RADC SOW. Key features that distinguish the CSC/SEA design are as follows:

1. Use of proven software and systems engineering concepts to produce a MAPSE design that provides a fully self-contained Ada development facility.

2. A MAPSE that provides a highly modular, easily rehostable system structure due to its machine-independent/virtual operating system support base.

3. A KAPSE environment that supports multiple Ada processes, as well as Ada tasking, that is fully extensible to multiprocessor host systems.

4. A KAPSE data base structure oriented toward providing full programming and project management support with an efficiency that meets large scale DoD programming needs.

5. Use of proven design techniques in the development of a performance oriented Ada compiler.

6. An Ada compiler design that minimizes memory requirements and maximizes rehostability.

7. An Ada compiler design that places paramount emphasis on the generation of efficient, highly optimized code coupled with ease of retargetability.

8. An integrated system for defining, manipulating, and managing configurations, versions and history attributes.

9. A configuration management system that provides features for maintaining both object grouping and object derivation.

10. A configuration manager that determines minimal operations sequences required both to update and to reconstruct objects.

11. A program library structure specifically optimized to provide fast, efficient, program linking features.

12. User-controllable line numbered text files that provide a consistent and stable interface between the Compiler, Editor and Debugger.

13. An efficient, rehostable and retargetable Debugger that uses the separate process technique to allow debugging of operational software with a minimum of interference.

The MAPSE is composed of three major components: KAPSE Data Base System (KDBS), KAPSE Framework (KFW), and MAPSE Tool Set.

The KDBS maintains and controls the use of all data objects and is the repository of all information used by MAPSE tools. It contains all data associated with the development of Ada programs and Ada program libraries.

It also provides project management and configuration management services in conjunction with tools from the Tool Set. The design of the KDBS is independent of the host environment. The mapping of the KDBS facilities into host facilities is provided by the KFW.

The KFW is the virtual operating system of the MAPSE. It provides run-time support facilities to supplement those provided by the KDBS, schedules Ada programs and tasks for execution, and handles all interfaces with the host environment.

The MAPSE Tool Set, consisting of the APSE Command Language Interpreter, Configuration Management System, Compiler, Linker, Editor, and Debugger, provides specific aids for the MAPSE user in the development of Ada software. Tools are designed to be as machine independent as possible. The Tool Set relies on the KDBS for data base management support and on the KFW for host system support. The facilities of the KDBS and KFW that are directly available to the MAPSE Tool Set and user programs are made visible through Ada package specifications, the collection of which is called the KAPSE virtual interface. This KAPSE virtual interface provides Ada specifications of all functions available to MAPSE tools and user programs to invoke other Ada programs, interact with the KDBS, and communicate with the KFW.

SECTION 2 - DESIGN PRINCIPLES

The objective of this preliminary design effort is the design of a program development and maintenance environment for embedded computer system projects involving Ada programs, with the intent of improving long-term cost effective software reliability. An integrated, comprehensive environment design facilitates the priorities for software quality in military embedded computer applications: reliability, performance, evolution, maintenance, and responsiveness to changing requirements. The system is designed to be rehostable and retargetable, providing flexibility in future applications. The MAPSE is designed to provide portability of the system itself as well as facilitating project portability.

2.1 MAPSE DESIGN GOALS

The MAPSE design goals incorporated in the CSC/SEA design include:

1. User-friendly and efficient interfaces
2. Logical entities and relationships to provide communication between the user and the MAPSE
3. Host logon facility extension into the KAPSE to improve user authentication and provide data privacy and security
4. Isolation of machine dependent interfaces in a single modular primary component, the KFW, dealing directly with host characteristics
5. A system designed to support the full range of Ada language features coupled with a highly efficient Ada compiler
6. Functional extensibility of the environment to facilitate MAPSE to APSE enhancements
7. A portable, retargetable, and readily maintainable environment.

2.2 STONEMAN DESIGN GUIDELINES

Specific design guidelines as described in STONEMAN include provision of an effective system development facility, MAPSE flexibility, and design of a practical system with priority given to user-friendliness. Each of these is described below.

2.2.1 Effective System Development Facility

In order to provide an effective system development facility, support is provided to projects throughout the software life cycle from requirements specification and design through implementation to long-term maintenance and modifications. The MAPSE supports the development of Ada programs by providing a state-of-the-art Ada Compiler that will handle full Ada as defined in the Reference Manual for the Ada Programming Language (July 1980), a set of program development tools, and all of those functions required by a project team, including project management control, documentation and recording, and long-term configuration and release control.

2.2.2 MAPSE Flexibility

MAPSE flexibility is enhanced by designing the system to be as portable as practicable, providing consistency across host environments, supporting the addition of other system tools and supporting the development of software for various embedded target computers. Portability is provided by writing the system in Ada and by identifying and isolating machine-dependent features of the MAPSE. Consistency across host environments is provided by designing a standard user interface and by providing host-like facilities within the MAPSE itself so that the capabilities of the system and the user access to these capabilities does not change from host to host. Through the provision of a standard interface, additional tools can be added to the MAPSE to allow the system to evolve into a full APSE. Applications of the MAPSE will involve developing software for embedded target computers. Thus, the design has considered the amount and type of target support necessary for these applications.

2.2.3 Practical System Design

In order for the MAPSE and the Ada language itself to be accepted by current users and adopted by future users, it must provide a practical system. The structure of the design is based on simple overall concepts that are straightforward to understand and use. Whenever possible, the design has

drawn on concepts that have been successfully implemented to reduce the risk involved in this system. The MAPSE has been designed to be reliable and robust, to protect itself from user and host system errors, and to provide meaningful diagnostic information to its users. Whenever possible, the concepts of the Ada language are used throughout the MAPSE. The MAPSE has been designed to provide adequate response/turn-around times for its users by considering efficiency issues in the design and by exploiting host system capacity and performance insofar as possible. The MAPSE system implemented from this design will be highly maintainable because it has been designed in a modular top-down fashion, is written in Ada, and is well documented. User documentation will be sufficiently detailed to enable the Government to maintain the system readily. In addition, for the first year after delivery of the system, maintenance will be provided by the CSC team.

2.2.4 User-Friendliness

High priority has been given to human engineering requirements in the design. The APSE Command Language (ACL) has been designed to provide commands that are both simple and flexible. Adequate response time will be provided and the MAPSE will be tolerant of user errors and provide meaningful diagnostic messages when errors occur. The ACL will support a broad spectrum of users from systems programmers to those with no programming experience. The MAPSE provides a well coordinated set of useful tools, with uniform intertool interfaces, standard communication capabilities, and a common data base that serves as the information source and data repository for all tools. The standard intertool interface can be exploited to facilitate the development and integration of new tools. Improvements, updates, and replacement of tools will be supported.

2.3 MAPSE CHARACTERISTICS

In order for the MAPSE to provide a usable programming environment it must demonstrate reliability, maintainability, and reasonable performance. Without any one of these characteristics, the user may find it difficult to use the MAPSE effectively.

2.3.1 Reliability

The MAPSE must be reliable. Whenever there is a switch from an existing system or operational mode, the reliability of the new system is of utmost

importance. The requirement for reliability of the MAPSE means that it does not promote errors and that it is tolerant of user errors. Upon catastrophic error, whether user or system generated, the MAPSE provides users with sufficient means to recover and be left in a well-defined state with minimal loss.

2.3.2 Maintainability

Several factors affect the maintainability of a large system such as the MAPSE. Perhaps the most important of these factors is that all components of the system are designed with a consistent philosophy. The MAPSE accommodates maintenance and upgrade of host systems. New releases of host operating systems may provide new features that would improve the MAPSE system. The maintainability of the MAPSE is supported by defining standard visible specifications of facilities provided.

2.3.3 Performance

The performance of the MAPSE depends on the host configuration, number of users, host performance, host/target interactions, as well as on the MAPSE design. Response time, which is the usual criterion for measuring interactive systems, varies according to the amount of processing required for a particular terminal input, the load on the MAPSE, and the load on the host. Reasonable response time is a prime requirement because it determines the usefulness of the MAPSE. The MAPSE has been carefully designed so as not to preclude its efficient implementation.

SECTION 3 - MAPSE SYSTEM DESIGN

The MAPSE system designed in this preliminary design effort meets all of the requirements as stated in the SOW. It will provide all of the data base support, interfaces to host facilities, user interfaces and tool interfaces necessary to satisfy requirements of the MAPSE tools and system users. All external interfaces are isolated, clearly identified and designed to facilitate portability of the system. Uniform protocol conventions for communication between users, tools, and the system have been established. The MAPSE software is designed to be modular and reusable; tools, such as the Linker, that may be required simultaneously by more than one user will be implemented as reentrant sharable code. MAPSE documentation will provide the necessary information to allow safe use of such code segments.

The CSC/SEA team has designed a system that meets not only the stated requirements of the SOW, but also those requirements derived from adherence to the guidance contained in the STONEMAN environment report.

The MAPSE system development has evolved from adherence to four major design principles:

1. The use of proven systems engineering and software design concepts to reduce risk.
2. The requirements to provide a fully self-contained Ada development facility to support both program development and program management.
3. The requirement for a highly modular design to support the evolution of the current systems design to a full APSE system.
4. The requirements for support of machine-independent virtual operating system structure to meet the goal of providing an easily rehostable MAPSE.

An overview of the major system components is illustrated in Figure 3-1. The basic design approach is to provide a portable programming support environment by developing a KAPSE and surrounding it with a minimal tool set

to form a MAPSE. The KAPSE components consist of the KDBS, which provides a machine-independent data base facility for users, and the KFW, which provides the host-dependent interface for the system as well as internal process scheduling and control. To provide consistency across host environments, the KFW provides a virtual operating system so that the user need not interface directly with the host. The MAPSE tools designed for this implementation of the MAPSE include: the APSE Command Language Interpreter (ACLI), which provides the user interface to the process and data base management facilities of the KAPSE; the MAPSE Editor, which provides basic text editing facilities for source programs and program documentation; the Ada Compiler, which supports full Ada and is designed to be rehostable and retargetable; the Linker, the Debugger, and the Configuration Manager, which works in conjunction with the KDBS to provide version, history, and configuration control.

The preliminary modular design approach for the MAPSE is based on past experience with the development of portable programming support environments and with effective programming support tools. This experience include the design and implementation of a portable JOVIAL programming support environment, called the Communications Software Development Package, which will be hosted on both the DEC System 20 and the Honeywell Multics System. Extensive experience with UNIX TM Multics, SHARE/7 and TOPS-20 operating systems has contributed to the selection and design of the support environment. The CSC Computer Sciences Timesharing System (CSTS) also contributed to the design of the MAPSE.

3.1 THE KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)

The components of the KAPSE are designed to provide the support necessary for the development and maintenance of Ada programs throughout the software life cycle. It provides the data base support, interfaces to host facilities, user communication interfaces, and tool communication interfaces in order to satisfy the requirements contained in the SOW. Host-independent interfaces are supplied for user-tool, tool-tool, and tool-KAPSE communication. These interfaces are supplied through the KAPSE virtual

interface, which provides the capability for the user to invoke any tool, for tools to invoke other tools, for communication to pass from user to tool and among tools in a controlled manner, and for communication with the KDBS and the KFW. User logon/logoff, initiate/terminate functions are provided through this interface by the KFW.

The KAPSE consists of two of the Computer Program Configuration Items (CPCIs) deliverable under this contract, the KDBS and the KFW.

Two key issues, one in each of these CPCIs, provided major design objectives that had to be met within the KAPSE.

1. The full ADA tasking capabilities must be extensible to multiprocessor host systems as well as multiprocessor targets. Ada tasking must also be compatible with the multi-user requirements for the MAPSE.

2. The data base structures must provide both full programming and project management support with an efficiency that meets large scale system development needs.

The design meets both of these objectives completely. The underlying structure of the KFW provides full apparent concurrency and is extensible to any multiprocessor host environment to provide real parallel processing should that feature be available. The data base provides a hierarchically structured logical data base with full facilities for object maintenance and manipulation in the software development environment. It also incorporates all the requisite facilities for project management control of that software development environment. As such, it provides an integrated system for defining, manipulating, and managing configurations, versions, and history attributes.

The interface between the KAPSE components and the MAPSE tools is described below.

3.1.1 The KAPSE Framework (KFW)

The goals of a portable MAPSE efficiently catering to multiple users requires that the traditional notion of a run-time support interface to the host environment for each user program be synthesized into the concept of a virtual operating system. The KFW is designed to fulfill this role in the MAPSE.

The KFW presents the facilities through which the user accesses the host operating system. These facilities are embodied in a virtualization of operating system services that provides for resource management, process scheduling, and servicing of user requests. The view of the KFW presented to users and to the MAPSE Tool Set will be consistent from implementation to implementation. The KFW will also provide the translation of the user requests from the virtual system to the host system. The KFW may execute on a bare machine or under an existing operating system, depending upon the implementation. In each instance, the KFW must interface directly with the host to provide the support for the canonical interface that is visible to the portable MAPSE components through the KAPSE virtual interface.

A principal objective of the KFW design is to optimize the coexistence and integration of the MAPSE and the underlying operating system. The MAPSE user's awareness of the host environment should be minimal or nonexistent, but the MAPSE, through the KFW, should exploit existing facilities, where appropriate, to maintain the required efficiency.

Standard terminal interface specifications and functions are provided through the KFW to facilitate the use of a variety of batch and interactive terminals and to ensure that machine-dependent interfaces do not affect the user. The KFW also provides the host interfaces required to support low-level I/O functions and basic data transfer and control functions. All host dependent computer programs necessary to implement the MAPSE system on the IBM and Interdata computers specified for delivery of the system will be specified and implemented as part of the KFW. Although these initial hosts are both uniprocessors, considerable attention has been given to the design of the KFW control functions so as to permit efficient implementation on multiprocessors.

An inherent property of the KFW is the ability to supervise and control the execution state of the MAPSE within the constraints imposed by the host environment. A fundamental requirement of this execution state is the servicing of multiple users or processes. Consequently, the design contemplates the existence of a privileged state in which only the KFW can execute and which has global rights to the MAPSE. In this state all MAPSE execution contexts may be accessed and controlled. In addition, where appropriate, the KFW executes in the context of a MAPSE process. A useful analogy is to consider the KFW as a consistent, systematic extension to the Ada Standard Environment. This extension specifies and implements the necessary functionality to comply with those Ada semantics that are dependent upon or relate to the execution environment. This functionality has commonly been identified as run-time support.

The Ada Run-Time Support Package, supported by both the KDBS and the KFW, provides for high-level I/O and the basic run-time facilities needed to execute Ada programs, such as the Ada Tasking Package. The Ada high-level I/O Package is supported by the KDBS and KFW. Facilities will be comparable to, or exceed, those typical in existing FORTRAN systems.

The KDBS and the KFW will both contain certain kernel functions that are not visible to the average user. These portions of the components are labelled KDBS Kernel and KFW Kernel, respectively. The KDBS Kernel will provide I/O support for the data base functions and will interface directly with the KFW Kernel, which provides the functions of an I/O dispatcher, a process administrator, an event monitor, and a context manager. The KFW Kernel will interface with the portions of the KFW that are machine-dependent and provide the direct interface with the host system.

3.1.2 The KAPSE Data Base System (KDBS)

The KDBS will present a logical data base structure to the user that will be hierarchical. It will be the logical repository of all MAPSE data and will provide the capability to create, delete, modify, store, retrieve, input, and output data base objects. It incorporates the requisite data base management facilities to control these manipulations. The KDBS provides

flexible storage facilities to all MAPSE tools and supports the creation and storage of Ada libraries in source form. The KDBS provides the capability to define, create, and manipulate categories, partitions, attributes, versions, and configurations of objects. In conjunction with the Configuration Management System, the KDBS controls access to data base objects based on version qualifier, attributes, and partition/configuration information. This capability is flexible to allow controls to be modified or redefined from project to project. The KDBS provides for archiving and for backup/restoration of data base objects in such a manner as to retain the integrity, consistency, and eventual availability of the objects. All KDBS functions will be implemented as callable Ada procedures or functions, and therefore will be available to Ada programmers.

All logical operations on objects, such as read, write, create, and delete, are performed by the KDBS. Object name expansion for use by tools is accomplished by the KDBS to ensure consistency among all MAPSE tools. When an object is opened, user access and usage rights are validated. Objects changed or created obey the KDBS policy for versioning.

3.1.3 KAPSE-to-MAPSE Interface

The KDBS and KFW interface with the MAPSE tools through the KAPSE virtual interface. Operations made available by the KAPSE permit the MAPSE tools access to the data base, and allow executing processes to initiate and communicate with other executing processes. This functional aspect of the virtual interface is designed with particular attention to completeness and consistency. Completeness ensures the portability of MAPSE programs as future KAPSE rehostings will find the set of virtual interface facilities sufficient. Consistency is important in order to facilitate a compact and efficient KAPSE implementation, and to present a uniform, straightforward virtual machine to the user.

As indicated above, the virtual interface can be viewed as the union of functional specifications for data base and process communication facilities. The functions visible to this interface consist of utility packages and the Ada Run-Time Support (RTS) Package, as illustrated in

Figure 3-1. The KDBS Utility Package provides access, attribute, archive, backup, and partition support to the MAPSE user. The KDBS also supports Ada I/O, which is contained in the Ada RTS Package. The KFW Interface Package makes available the KFW facilities for process initiations and control. The task management functions of the Ada RTS are provided by the KFW.

Figure 3-2 illustrates the KAPSE/MAPSE interfaces as described by this design. The tools communicate with each other and with the KAPSE through the KAPSE virtual interface, which contains the specifications of the Ada RTS Package, the KDBS Utility Package, and the KFW Interface Package. The KDBS Utility Package relies on the KDBS Kernel to supply the needed I/O functions. The KDBS Kernel, in turn, receives I/O services from the host through the KFW Kernel. Similarly, requests for KFW services are channeled through the Ada RTS Package or the KFW Interface Package to the KFW Kernel for service. Only the KAPSE virtual interface is visible to the user. The host system only sees the host interface with the KFW Kernel. All other interfaces are internal to the MAPSE system. Details of the operation and functions of all of these KAPSE components are given in the appropriate sections of this interim technical report and in the corresponding B5 specifications.

3.2 THE MAPSE TOOL SET

The KAPSE is enhanced by the addition of the six MAPSE tools: the ACLI, the Configuration Management System, the Ada Compiler, the Linker, the Editor, and the Debugger. This MAPSE tool set comprises the remaining six CPCIs to be delivered under this contract.

3.2.1 APSE Command Language Interpreter (ACLI)

The ACLI provides the most common and standard user interface to the MAPSE. Its basic function is to interpret ACL commands, which are usually directives to execute an Ada program as a MAPSE process. The ACL combines command language and programming language constructs to support a broad spectrum of users. Although it is possible to construct complex and powerful command files, the ACL is designed primarily to provide simple, straightforward commands. Capabilities not available in the ACL can be obtained by the user through Ada programs.

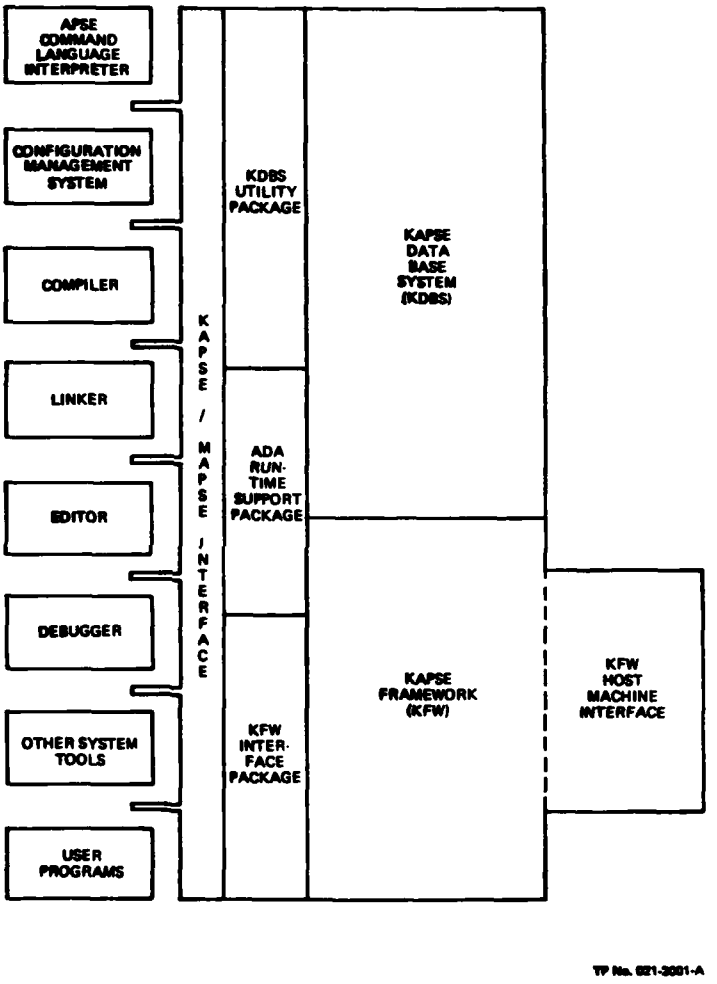


Figure 3-1. MAPSE System Diagram

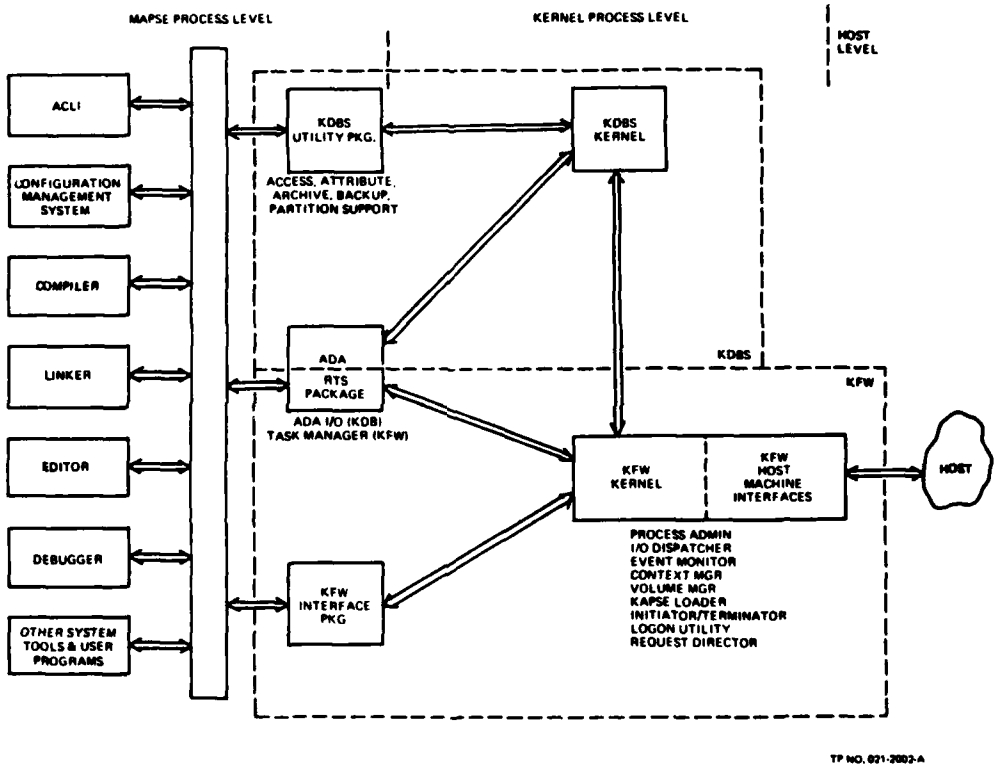


Figure 3-2. KAPSE/MAPSE Interfaces

The purpose of the ACLI is to provide a user-friendly, efficient means of initiating and communicating with processes. This requirement precludes the use of full Ada as the Command Language. It is anticipated that many users of the Command Language will not be Ada programmers. Moreover, it is unlikely that an Ada interpreter would be practical. The goal, therefore, is to provide sufficient functionality without undue syntactic overhead. The ACL has been designed to exploit Ada constructs whenever possible, and to extend the capabilities needed for a simple, efficient, user-friendly command language. The ACLI is designed to be usable by anyone, it has no declarations and, for common operations, no knowledge of Ada is necessary. The ACL uses the standard Ada character set. Ada-like statements function as they would in an Ada program, given the restriction that this is an interpreted, not compiled language.

The ACLI provides the capability, through the KAPSE virtual interface, for users and tools to invoke other Ada programs. Uniform formats for user commands and tool invocation are provided through the ACLI. In addition, the format for similar commands used by different tools are uniform and consistent throughout the MAPSE.

A number of essential command utilities are included to augment the language features of the ACLI. In particular, these include report generation facilities to provide the following reports: configuration composition report, attribute report, partition report, attribute select report, and summary reports based on combinations of attribute, partition, configuration, or version qualifiers.

3.2.2 Configuration Management System (CMS)

The Configuration Management System (CMS), in conjunction with the KDBS and the ACLI, provides facilities to support project configuration management. The CMS was designed against two specific requirements for object control:

1. The CMS is required to provide features for maintaining both object grouping and object derivation.

2. The configuration manager provides features for determining minimal operations sequences required to both update and reconstruct the objects.

The CMS is part of a unified system to handle version, history and configuration management. The CMS will also aid in project management and assist programmers in defining configurations.

The CMS processes and maintains configurations objects (COs). A CO defines a set of data base objects along with a set of rules, written in ACL, describing how objects are to be derived from other objects.

COs are used by the CMS to provide for object updating and object reconstruction. The updating function will bring objects in the configuration up-to-date with respect to each other. This facility easily enables the definition and enforcement of compilation order rules. The reconstruction function allows any previous version of an object to be reconstructed. This facility has been carefully integrated with object history attributes to achieve both space and time efficiency in reconstruction.

3.2.3 Ada Compiler

The Ada Compiler has as its primary purpose the compilation of Ada programs. In addition, it facilitates source formatting, statistics collection and reporting, program flow description, Ada program library creation and maintenance, diagnostic reporting, listing generation, compilation order validation, and internal compiler debugging. These features, many of which could be implemented as separate tools, are included as modules within the Compiler to provide a more useful facility to the initial MAPSE user. The modularity of the Compiler will enable these to be user options.

The key design goals of the Ada compiler are as follows:

1. Use of proven design techniques in the development of a performance oriented Ada compiler.

2. An Ada compiler design that minimizes memory requirements and maximizes rehostability.

3. An Ada compiler design that places paramount emphasis on the generation of efficient, highly optimized code coupled with ease of retargetability.

The Ada Compiler processes Ada source programs and produces efficient equivalent programs in the form of object modules. It is designed for batch, remote batch, and on-line use and processes the full Ada language. Language-defined pragmas and other pragmas used to convey information to the Compiler are specified. The design is modular to minimize host resources required. The Compiler is designed to tailor its memory utilization to accommodate source programs of different sizes. While the Compiler is designed to be easily rehosted and retargeted, object code efficiency is of paramount importance. Code generator phases for different target machines are designed to be separable. The machine-independent parts of the Compiler will be parameterized. The front-end will be able to interface with a variety of back ends, which contain the machine-dependent characteristics.

The Compiler design minimizes the host computer system resources required for a compilation and permits utilization of available memory to improve compilation speed and increase internal compiler table limits.

An innovative concept to enhance the optimization effectiveness is to preprocess the Intermediate Language (IL) produced by the front-end to incorporate machine-dependencies into the program representation. Included in these dependencies are such information as register-versus-stack guidance, number of registers, calling convention expansions, addressing requirements, loop control orientation, etc. The intent is to supplement the IL with information that orients the global optimization toward the selected target while retaining the machine-independence of the major optimizing phases.

Optimization occurs at the user's option through the use of language pragmas. Global optimizations to be provided include common subexpression elimination, loop optimizations, dead code and variable elimination, constant and value folding, code motion, and inline substitition. Local optimizations provided include order of expression evaluation, peephole optimizations, subscript linearizations, Boolean optimizations, constant arithmetic and conversions. Ada-related optimizations to be performed include exception checking, discriminant validation, space reclamation and management, and generic subprogram combining. Machine-oriented optimizations include path merging, pattern matching, register allocation, full instruction usage, efficient calling sequences, and addressing mode optimizations.

The Compiler performs extensive error checking and reports errors in a meaningful manner to the user. Error severities will be reported as notes, warnings, errors, serious errors, and fatal errors. Compilation will continue unless the error is serious or fatal, but the object code produced may be unreliable. Error messages will contain an error identifier, severity code, and descriptive text indicating the meaning of the error. All syntactic and semantic errors as identified by language constructs are to be detected. Capacity errors will also be indicated. A complete list of all error messages capable of being generated by the Compiler will appear in the Users Manual, which will be delivered at the end of Phase II.

As the only component of the MAPSE Tool Set that is required to parse and analyze Ada source programs, the Compiler is in a position to collect information about the program and to perform as a simple, inexpensive, compilation by-product certain functions that would otherwise require substantial and largely redundant software. Furthermore, because the Compiler is one of the programs that must be used during program development, it may perform certain administrative functions unconditionally, and even unknowingly to users. Performed otherwise, these functions would require an additional level of control or conventions to achieve.

3.2.4 Linker

Performance considerations are a major concern in the Linker design. The Linker requires multipass access to the programs being linked. The programs must be accessed at least once to locate the descriptive information about the program, satisfy all references, and compute the allocation. They must be accessed again to perform the reference resolution, address relocation, and creation of the new program image. Design decisions, particularly with respect to program libraries, have been made to minimize the overhead normally incurred during the opening and closing of the relocatable objects.

The present design utilizes the relocatable object format as the executable format. To minimize the requirement for dynamic relocation at load time, the Linker optionally biases addresses to eliminate this relocation where possible.

The Linker combines two or more relocatable object modules in standard object module format and produces a standard relocatable object module. It will perform external name resolution and location counter relocation, space allocation, memory map generation, and will validate version and compilation order for the objects to be linked. The Linker is designed to operate in the environment of the MAPSE as a machine-independent tool.

3.2.5 Editor

The Editor will provide line and string insertion, replacement, deletion, copying, and moving. Its primary purpose is to support text object creation, modification, and browsing facilities. The Editor will recognize Ada constructs, interpret sophisticated commands, and use simple commands for frequent functions. It has been designed to provide primitive word processing capabilities to support the development of program documentation, in addition to its function in source editing.

This design imposes no size constraints on the object being created. Modern text search algorithms are utilized to perform the Editor commands efficiently.

A design objective is to build an understanding of Ada symbols into the Editor, although the Editor need not recognize full Ada syntax. This allows the location or replacement of strings of symbols regardless of spaces and line boundaries. This facility prevents the inadvertent location of a series of characters that are a part of a larger token. For example, optionally, a string may be searched for as an entire identifier, and would not be found if its occurrence is only as part of a larger identifier, comment, or string literal.

To promote nicely formatted Ada programs, the Editor provides features to support Ada declaration and statement-structure indenting and tab character facilities.

Because the Editor will frequently be used for the entry and modification of software documentation, the Editor provides a capability to establish margins, to set listing page length, to allow line filling and justification, to set up and reference file markers, and to locate and replace on a word basis rather than on an unspecified context basis.

Often during the production of a large programming system and the accompanying documentation, there is a need to combine or partition the associated text objects. The Editor supports this requirement.

3.2.6 Debugger

The MAPSE Debugger will provide symbolic, interactive debugging support to the user. It will permit breakpoints and program interruption, symbolic and machine level inspection and modification. The Debugger provides symbolic and relative program dumps and traces subprogram calls and statements. It will interface with target simulators to aid in debugging programs written for a target machine.

Performance considerations are a major concern of the Debugger design. The technique selected to facilitate program control uses instruction implantation rather than interpretation. To prevent a space impact due to an increased memory space requirement for containing the tables, portions of the debug tables will only be brought into memory as required.

Although the normal mode of program checkout is to load a program, perform some initial setup of breakpoints or implanted commands, and then initiate interactive execution, an important requirement is to permit interruption of an already executing program showing signs of aberrant behavior. The most difficult bugs are those occurring in large programs after substantial processing has been performed. The requirement for embedded debugging code can never be completely anticipated and apparently thorough test cases seldom reveal problems of timing, space management and the interactions of optimization and all language features.

3.3 SUMMARY

This introduction has provided an overview of the CSC/SEA design for the Ada Integrated Environment. The following sections address each major component of the system individually. Each of the following sections is presented from the viewpoint of the component itself. Interfaces to the rest of the system, the design principles that guided its development, the functional description of the component, and the rationale for the design decisions, are presented for each of the system elements.

INTERIM TECHNICAL REPORT

TAB 1

KAPSE FRAMEWORK

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the KFW preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements on which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The KFW provides the administrative, control, and resource management services that are necessary for the MAPSE to support the execution of multiple programs interacting with a shared data base. These services are provided through a canonical interface to a virtual operating system structure that has well defined interfaces to those host system facilities that are normally defined and provided by various host execution domains.

1.2 DESIGN PRINCIPLES

The KFW is designed to comply with a set of principles considered essential in attaining a prototype MAPSE that conforms to the spirit of the STONEMAN Report and the requirements of the SOW. These principles are the rationale for the decisions that have motivated the design model for the KFW. In addition, the design is formulated to complement the proven software technology that is synthesized in the Ada language. As a result, the development of the detailed design for the KFW can be expressed using those Ada constructs suited to preparing formal specifications for complex systems.

The principles that have influenced the design are:

1. The portability of MAPSE components
2. The support for a multi-user environment
3. The adaptability of the design to diverse hosts
4. Concise interfaces
5. The effective decomposition and execution of the defined KFW functionality

6. The exploitation of multiprocessor architectures.

Figure 1-1 illustrates where these design principles are best exemplified in the KFW design.

1.2.1 MAPSE Portability

The requirement to ensure a high level of portability for all MAPSE components through the encapsulation of the host hardware/software dependencies within the functional domains of the KFW is regarded as the original precept for the existence of the KFW. All MAPSE components are perceived as Ada programs that are provided with a canonical interface to the host environment. This interface is made available through Ada packages that constitute the visible specification of the KFW. The use of each package is restricted by the inherent access control features for Ada Program Libraries that are maintained in the KAPSE data base. By strict adherence to this principle, a measure of portability, in the KFW, is realized as a desirable side effect.

1.2.2 Multi-user Environment

The notion of the MAPSE mandates that it be responsive to multiple users. Traditionally, this results in increased complexity in the design specification. Recognition of the multi-user requirements early in a design specification is critical in order to develop the necessary parallel functionality. Therefore the KFW is designed to support such parallelism within the MAPSE. Because of the early acceptance of this design principle, both the ACL and the KDBS take advantage of the enhanced multi-user interaction capabilities.

This design principle is incorporated in the KFW in concert with the concurrency features that are defined as an intrinsic capability of an Ada program. Consequently, the resulting design specification defines the semantics for the execution of multiple Ada main programs. This is an area not within the scope of the current language specification, but absolutely essential for the establishment of large-scale software development systems.

	KFW Functional Domains										
Design Principle	KAPSE Initiator	Logon Utility	Request Director	KAPSE Terminator	Process Admstr	Task Manager	Context Manager	Event Monitor	Volume Manager	I/O Dispatcher	KAPSE Loader
Portability of MAPSE	X	X	X	X	X	X	X	X	X	X	X
Multiuser Environment		X			X		X	X		X	
Adaptability to Host			X				X	X	X	X	X
Concise Interfaces					X	X	X	X	X		
Effective Decomposition	X	X		X	X	X	X	X	X	X	X
Multiprocessor Architectures					X	X					

Figure 1-1. Application of Design Principles

1.2.3 Diverse Host Adaptability

The two initial host environments are the IBM VM/370 and Interdata OS/32. This predicates that a design principle for diverse host adaptability be followed in the KFW. The VM/370 provides to the KFW a low level interface while the OS/32 provides a conventional multiprogramming operating system interface. Therefore, for economic and viable implementations on the immediate hosts, it is essential that host adaptability of the KFW be introduced into the design so that reliance on specific nongeneric host features be minimized without compromising the operational performance of the MAPSE. Adherence to this principle increases the scope of the initial design because functional domains must accommodate a complex mapping of the canonical interface into either hardware or software materializations for host execution.

1.2.4 Concise Interfaces

A major concern in the design of a complex network of service programs is to present the encapsulated facilities through concise interfaces that are simple and well understood. The design principle derived from this concern has influenced the specification of the KFW interfaces. The anticipated goal to formally specify and verify programs in the MAPSE emphasizes the need to limit the KFW facilities to the essential requirements of the MAPSE. These specified facilities use reliable techniques that do not preclude the formal specification of the KFW or other MAPSE components.

In the KFW design this requirement is balanced so as to comply with other important MAPSE objectives. This is typified by those interfaces that are used in the object code generated by the Ada Compiler. Minimization of KFW facilities is carefully orchestrated to support requirements for efficient and straightforward object code generation.

1.2.5 Effective Decomposition

The Ada language embodies a principle for the effective decomposition of large, complex programs through the package and separate compilation features. This principle is perpetuated in the design of the KFW by

isolating and specifying it by functional domain, thereby logically extending the concept of component software technology beyond the MAPSE tool set.

Compliance with the previous design principles of host adaptability and concise interfaces combined with effective decomposition leads to the analogy that the KFW represents a virtual (generic) operating system included in the Ada Standard Environment that is instantiated for each host system. The embodiment and refinement of this concept in the MAPSE design is an ambitious but achievable goal necessitated by the requirement for maximal portability.

Recent experiences in Ada compiler technology to develop packaged compiler components clearly demonstrate that pursuit of this principle is a consistent progression in the propagation of economic software.

In addition, application of this design principle is especially germane to the MAPSE, where real-time software can be initially developed prior to final testing and production execution in an embedded target environment. Effective decomposition permits ready identification and modification of the required functional domains for the embedded software for real-time execution in the target environment.

Finally the effective decomposition of the KFW is considered vital when combined with the application of the following design principle relating to multiprocessors. Each functional domain of the KFW may, in the future, be conceived as potentially executing on different processors.

1.2.6 Multiprocessor Architectures

The base architectures for the initial hosts are uniprocessors enforcing serialized execution of the MAPSE. Awareness of possible future hosts that provide for parallel execution of the MAPSE require that another design principle be adopted. This principle as applied in the KFW design ensures that when host multiprocessing capabilities are available they are not automatically precluded by a restrictive design geared to interleaved serial execution of the MAPSE. Rather, the KFW is designed so that multiprocessing opportunities can be exploited. This principle is clearly manifested in the

design of the interface between the Task Manager and Process Administrator. The importance of this principle is reinforced in those embedded target environments where multiprocessors are not uncommon. The availability of such a host KFW enables a target KFW subset to be ported without a major design overhaul. In addition, target software developed on the host is already debugged and is ported with a minimal change.

SECTION 2 - BACKGROUND

This section provides the background information on which the design of the KFW is based. The requirements for a portable environment have imposed on the KFW the necessity to provide to the MAPSE a virtual operating system so that the other MAPSE components are not required to have detailed knowledge of the underlying host operating system. In addition to providing this host interface, the KFW also presents a standard interface to the rest of the MAPSE system through the KVI. Machine-dependent and machine-independent portions of the KFW are separated and clearly identified to facilitate rehosting the MAPSE.

2.1 PREVIOUS WORK

Many of the design decisions which have been made in this preliminary design effort are the result of CSC's experience with another portable programming support environment, called the Communications Software Development Package (CSDP) which is being developed for RADC. The CSDP is a JOVIAL programming support environment and is being implemented on a DEC System 20. This system will be rehosted to the Honeywell Multics system at RADC in late 1981. The scope of the contract includes providing a portable user interface, a program support library, a set of support tools, a tool manager, and a machine-dependent interface. The design of the machine dependent interface has provided background and understanding in the design of the KFW. In addition, ongoing technical support to the U.S. Navy's Automatic Electronic Guided Intercept System (AEGIS) program for the Share/7 CMS-2Y programming environment executing on a UNIVAC AN/UYK-7 multiprocessor has provided practical insight into developing embedded software.

Finally, recent work undertaken at Stanford University to develop Ada-M a language for specifying and implementing multiprocessing supervisors has been reviewed. From this review it is expected that the proposed language can be used to provide a formal specification of the KFW design that will provide a reliable basis for implementation.

2.2 RELEVANT DOCUMENTS

1. Requirements for Ada Programming Support Environments "STONEMAN", February 1980.

2. Fisher, David A., "Design Issues for Ada Program Support Environments," Science Applications, Inc. SAI-81-269-NA, October 1980.

3. Ada-M, An Ada-based Medium-Level Language for Multi-processing, developed for DARPA, under Contract No. MDA 903-80-C-0159, March 1981.

4. Ada Support System Study, Phases 1, 2, 3, 4 developed for United Kingdom Ministry of Defence, Systems Designers, Ltd., Software Sciences, Ltd., 1979-80.

5. Communications Software Development Package (CSDP) System/Subsystem Specification, developed by Computer Sciences Corporation for RADC under Contract No: F30602-79-C-0051, July 1980.

6. Allshouse, Richard A., David T. McClellan, Gelbert Prine (CSC) and Cpt. Clair Rolla (RADC), CSDP as an Ada Environment, Novemeber 1979.

7. Reference Manual for the Ada Programming Language, July 1980.

8. Perkin Elmer Interdata Division, OS/32 Programmer's Reference Manual, S29-613R02,

9. Virtual Machine Facility/370, Features Supplement, GC20-1757-2.

10. CSDP; Functional Description; developed by Computer Sciences Corporation under Contract No. F30602-79-C-0051, October 1979.

11. Cowan, George, A General Structure for Resource Management in a Computer Network, PhD Thesis, University of Wisconsin, 1975.

SECTION 3 - FUNCTIONAL DESCRIPTION

The following section provides a general overview of the KFW interfaces within the MAPSE and describes the design tradeoffs performed during the KFW preliminary design.

3.1 INTRODUCTION

This paragraph dicusses the system interfaces and functional design capabilities for this system element in terms of the design principles outlined in Paragraph 1.2.

3.2 SYSTEM INTERFACES

The services of the KFW provided to the other components of the MAPSE are encapsulated in the MAPSE-KFW interface services of the KVI. These services may be executed in the calling execution state or in the privileged execution state, depending upon the function to be performed.

Figure 3-1 illustrates the MAPSE system interfaces. The bold vertical lines identify the interfaces that comprise the virtual operating system encapsulated by the KFW in the Kernel Process. These interfaces may be used by any MAPSE process, while the interfaces depicted as faint broken lines identify the interfaces supplied to the KDBS and for the support of Ada tasks.

3.3 FUNCTIONAL CAPABILITIES

Ada user programs and MAPSE tools execute in a nonprivileged execution state. The KFW considers these execution domains to be MAPSE processes. The functional domains of the KFW and KDBS that execute in a privileged execution state constitute the Kernel Process of the MAPSE. In order to support this functional dichotomy of the KFW and KDBS between a MAPSE process and the Kernel Process, the KFW supplies an interface that enables a MAPSE process to request a service provided by the Kernel Process. Any KDBS or KFW Kernel service that is requested by a MAPSE process is connected to the Request Director in the Kernel for the service to be recognized and routed to the appropriate Kernel function. The nature of the interface to

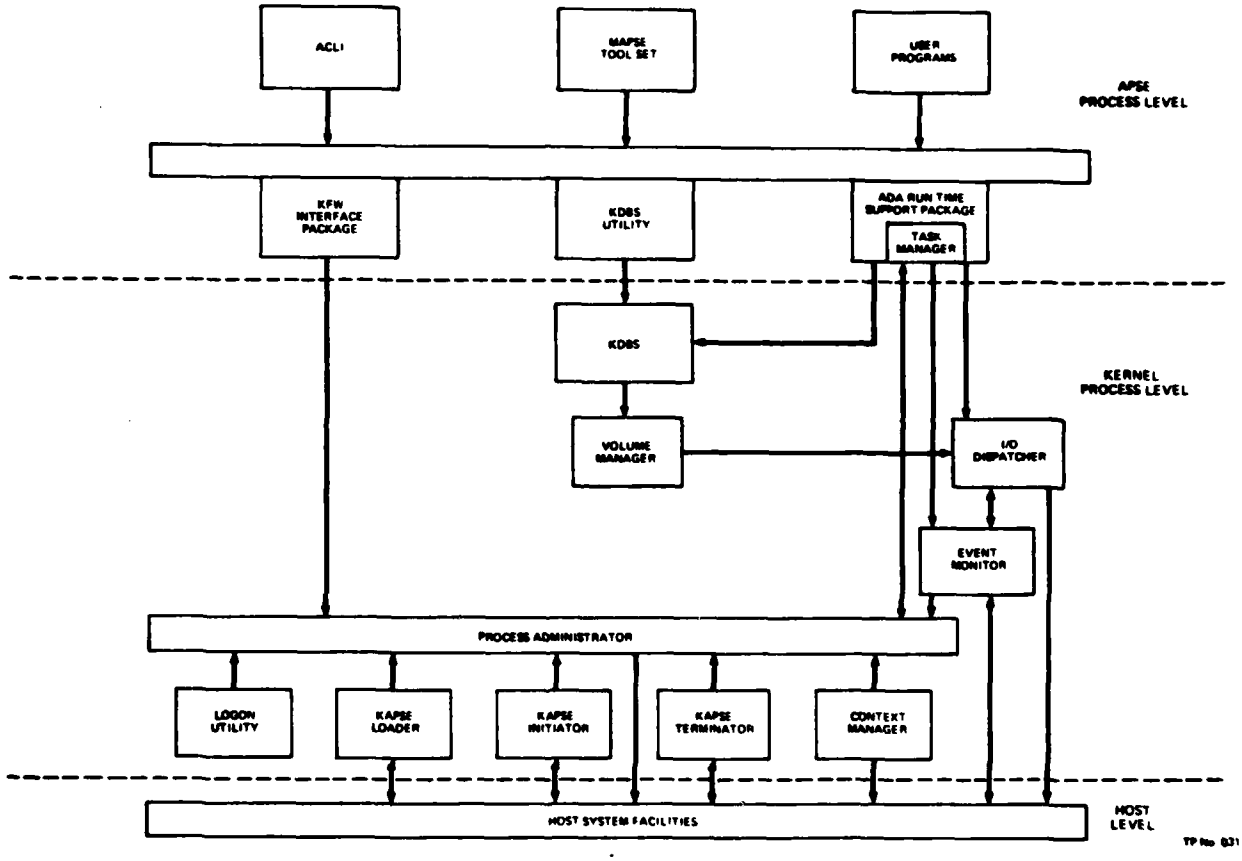


Figure 3-1. MAPSE System Interfaces

the Request Director depends upon the host system facility available for communicating between execution states.

The KFW is designed to include the Task Manager functional domain as an extension to a MAPSE process. All other KFW functional domains are included in the Kernel or execute as MAPSE processes.

The following sections describe the functional capabilities of the KFW in terms of operating system services, data base interface, communications services, and execution services. In addition, Paragraph 3.3.5 addresses each specific functional area of the KAPSE KFW design.

3.3.1 Operating System Services

The KFW services are visible to the other MAPSE components through the KVI, thereby enabling the other components to be designed with a minimum knowledge of the host environment. The interface is designed to specify the functionality that is usually offered in an operating system. Therefore, when the host execution domain includes an operating system, as in one of the initial implementations, the KFW services are derived from existing facilities to avoid duplication of or interference with a system facility.

The KFW design is based upon the availability of fundamental equipment that is a direct consequence of the system facilities provided by the IBM VM/370 and the Interdata OS/32 operating system. If this equipment were not available, the dependent KFW service would be provided in a degraded state. The following paragraphs identify the critical equipment characteristics that are beneficial to the KFW and the performance of the MAPSE. The equipment characteristics pertain to the processing architecture of the execution states in which the KFW resides.

The KFW interface is designed to comply with the requirements of the Ada language. In those instances where the language semantics are to be defined by implementation considerations, the KFW functionality is designed so that a minimum of constraints are imposed in exploiting the host execution domain. This results in the designed functionality being restricted when the host execution domain does not supply the foundation facility (such as multiprocessing).

An execution state that possesses properties not necessary for normal MAPSE processing is required. Properties that are associated with this state include the ability to reference any address domain defined within the MAPSE and the ability to use system facilities that have restricted visibility for controlled utilization.

The concurrent existence of execution states is required to allow the KFW to provide the parallelism implicit in the facilities it offers. This requirement has ramifications on both the hardware and software that constitute the system facilities. When this support is not available, the KFW parallel functionality is serialized and presents only logical parallelism to MAPSE programs.

Communication between a nonprivileged state and a privileged state is required. The KFW organizes its functionality in both states using the intercommunication between the states achieved through host system facilities.

The creation of a dynamic address domain that may be referenced by a nonprivileged state is required. The KFW is designed to exploit this facility to achieve economical information transfer between the privileged and nonprivileged states of the MAPSE. Ideally, a nonprivileged state must be restricted from referencing the dynamic address domain of a concurrently executing nonprivileged state.

The definition of shared execution domains that may be referenced by all nonprivileged states is required. The KFW is designed to exploit this facility so that KFW functions, where possible, may be performed in the nonprivileged execution states.

Because the KAPSE initiates actions that may be performed asynchronously in the host environment through the KFW, the KFW is designed to monitor completion of these actions by establishing an event protocol with the host environment. This protocol relies on the system facilities notifying the KFW that a requested action has been performed, so that the result of the action may be analyzed prior to the resumption of the execution state initiating the action.

3.3.2 Data Base Interfaces

A primary design objective is the integration of the KDB into the host environment to ensure the portability of the KAPSE functionality that maintains the logical entities within the data base. A consequence of this objective is that the KFW is designed to use system facilities to materialize logical entities of the KDB into items that are acceptable for storage on devices in the host environment. In addition, the KFW design provides those facilities required by the MAPSE to communicate interactively with terminal devices that are configured in the host hardware suite. Where feasible, the KFW relies on the system facilities to provide this functionality.

The KDB physically resides on devices in the host environment. These devices may be classified as logical or physical devices depending on the nature of the host system facilities. The entities (objects) comprised in the KDB are accessed by an Ada program without regard for this classification through the functionality provided by the KFW.

Logical devices are characteristically typified as files in the host environment and are manipulated through the services of a file management system. The KFW is designed to provide an interface to these services thereby enabling a straightforward correspondence between data base objects and files to be established and maintained. To ensure that the correspondence may be implemented using different file management systems, the KFW relies only on those services that are traditionally conceived as being generic to most host systems.

Physical devices are defined to include those devices configured in the host hardware suite on which data may be recorded and subsequently retrieved. The KFW is designed to provide an interface to these devices as required to support those data base objects that have been designated as devices for manipulation by an Ada program. The KFW relies on the availability of device handlers in the host system facilities so that the correspondence between a data base object and a device may be established and maintained in a manner consistent with that of a data base object and a file.

3.3.3 Communications Services

An important requirement in the utilization of I/O by the KFW is that in those host environments that do not include logical devices, the system facilities for the physical devices provide sufficient functionality for the KFW to support its specified interface.

The interaction between a MAPSE user and the MAPSE is enabled through the communication devices configured in the host hardware suite. The nature of the communication devices determines the style of user interaction that is available, batch, interactive, or both.

When console or terminal communication devices are configured, the KFW is designed to provide the functionality necessary for a user to converse with the ACL. The host system facilities for handling communication devices are used by the KFW to implement an interface that is responsive to the needs of all MAPSE tools that may establish a dialogue with a user.

A consistent user communication interface to the MAPSE requires that the KFW incorporate in its design a standard line editing protocol for console or terminal input. Host system facilities, while providing services for reading and writing characters to communication devices, are unlikely to conform to this protocol. Therefore, the system facilities must permit the KFW to implement the necessary functionality to support the editing of input characters without interference. A critical requirement is that the defined MAPSE break-in or attention signal be discernible by the KFW so that a user may be connected initially to the MAPSE Logon Utility, or for a user to terminate a current execution state in the MAPSE.

When non-interactive communication devices, such as card or paper tape devices, are configured, the KFW is designed to provide conventional batch operation by directing the device to the ACLI. Again, the host system facilities for handling these devices are used by the KFW.

The KFW is designed to support a variety of non-standard I/O requirements. These requirements arise from an Ada program and from the KFW itself.

Through the KFW, an Ada program is provided the functionality to connect to a device that is not in the prescribed host hardware suite. In this instance, the host system facilities must enable the KFW to have control of the I/O channel for the device so that the KFW may receive and send instructions or data from the Ada program to the device or device controller.

3.3.4 Execution Services

Other nonstandard inputs required by an Ada program are specific entry interrupts and clock data. The KFW is designed to field the interrupts and read the clock through the host system facilities. Similar interrupt and timer services are required by the KFW to detect the termination of asynchronous events that it may have initiated in the host environment. For example, the completion of a MAPSE I/O operation is recognized by its termination interrupt being made available to the KFW through the host system facilities.

The execution domain of the MAPSE consists of the Kernel Process and one or more MAPSE processes. The Kernel Process executes in a privileged execution state while the MAPSE processes execute in a non-privileged execution state. The Process Administrator is the part of the Kernel Process that is designed to coordinate and schedule the MAPSE execution domain so that it may coexist with other execution domains in the host environment. The host system facilities are used by the Process Administrator as necessary to ensure the efficient, economical execution of a process in the host environment.

A MAPSE process may invoke the execution of another MAPSE process through the Process Administrator. After invocation, the calling and called processes are candidates for execution. In order to comply with the semantics of the ACL, parameters may be passed between the calling and called process.

A consequence of the Ada task semantics is for a MAPSE process to synchronize the execution of different code domains (tasks) within the load object. The Process Administrator recognizes this requirement by maintaining the scheduling of a MAPSE process to be consistent with the task synchronization specified in the MAPSE process.

The Process Administrator is designed to facilitate the physical parallel execution of processes where the host system facilities support multiprocessors in the host hardware suite. When such facilities are not available the Process Administrator implements conventional logical parallel (interleaved) execution of processes.

In host environments that provide system facilities that preclude the Process Administrator's direct control over the scheduling of process execution, the Process Administrator relinquishes final scheduling of process execution to the host environment.

3.3.5 KFW Functional Areas

Each of the following eleven subsections describes the KFW function and its design goals in terms of the design principles stated in section 1.2.

3.3.5.1 KAPSE Initiator

The KAPSE Initiator receives control when the KFW Kernel Process is loaded for execution in the host environment. It prepares the MAPSE for process execution and starts execution of the Logon Utility. The KAPSE Initiator relies on the facilities provided by the Process Administrator and Context Manager and the information contained in the environment system parameters. As a result, it is a KFW functional domain that is designed to be portable when rehosting the MAPSE.

3.3.5.2 Logon Utility

The Logon Utility permits a user access to the MAPSE through an authentication protocol. For each authenticated user, an instantiation of the ACLI is started. The Logon Utility is a MAPSE process that relies on the KFW interfaces available through the virtual interface and is therefore portable. The design of the Logon Utility is specifically oriented to servicing a multi-user environment through the ACLI instantiation mechanism that it supports for each user communication device configured in the environment system parameters.

3.3.5.3 Request Director

The Request Director provides the KFW functionality through which Kernel Process facilities defined in the KAPSE virtual interface may be performed. It is designed to reconcile the transition mechanisms between the nonprivileged and privileged execution states. As a result, the Request Director executes as a part of a MAPSE process and in the Kernel Process. In those base architectures that feature comprehensive execution state transition facilities, the MAPSE process execution requirements are minimal.

3.3.5.4 KAPSE Terminator

The KAPSE Terminator receives control when the MAPSE is to be shutdown in an orderly fashion and the Kernel Process removed from the host environment. The KAPSE Terminator relies on the facilities provided by the Process Administrator and Context Manager. Consequently, it is designed to be portable when rehosting the MAPSE.

3.3.5.5 Process Administrator

The Process Administrator schedules MAPSE process execution through a concise interface made available to MAPSE processes. This interface is designed to enable the Task Manager to be specified as a portable functional domain within the KFW. In addition, the Process Administrator provides the necessary functionality for the MAPSE to be a multi-user environment. The multi-user environment is achieved by defining a concept of multiple executing Ada main programs. When an executing main program is an ACLI process connected to a user communication device all the requirements for maintaining a comprehensive multi-user environment are satisfied.

An intrinsic design feature of the Process Administrator is the recognition of potential multiprocessor environments. It is designed to exploit multiprocessor opportunities when available in its interface to the host system. Process executions are activated in accordance with the environment system parameters and multiple activations for the same process may be forwarded for host execution. In these instances, the Process Administrator uses the Ada task scheduling data maintained by the Task Manager. Task execution is viewed as simply different executions of the same process where

process identification for a thread of control is the Process Control Block (PCB) and a Task Control Block, the union of which constitute a PCB instantiation.

The scheduling and activation of processes result in the interleaved execution of tasks from different processes as compliant with any priorities that are to be maintained. If multiprocessing is available, these process executions may be performed in parallel.

3.3.5.6 Task Manager

The Task Manager maintains the task control and synchronization information required to support Ada task objects enclosed by a MAPSE process. It is designed to provide a concise but straightforward interface in order for the Ada compiler to generate efficient code for the tasking constructs of the language. The use of the Process Administrator interface makes the Task Manager portable. The Task Manager is designed to execute in a multi-user environment and is specified as an extension of a MAPSE process that resides in the shared execution domain.

Because the Task Manager relinquishes the actual scheduling of tasks to the Process Administrator, parallel execution of tasks is achieved by default in multiprocessor environments.

3.3.5.7 Context Manager

The Context Manager establishes and maintains the execution context requirements necessary to support a dynamic multi-user environment. It is designed to provide a concise portable interface that enables an executing MAPSE process to acquire and release storage by modifying its program context map. In addition, debugging facilities are supported by the interface, and allow the execution context of one MAPSE process to be dynamically changed by another concurrently executing process (such as the Debugger).

The facilities available through the Context Manager are designed to be compatible with and achievable in different host systems. Using the Context Manager, the Kernel Process prepares the initial state of the MAPSE for the

execution of multiple processes. The design provides the required interfaces to build and access the execution domain for the shared KFW and KDBS facilities necessary to efficiently support concurrently executing processes. These shared facilities depend on the Context Manager to acquire and protect dynamic address domains for storing private data.

3.3.5.8 Event Monitor

The Event Monitor maintains control of asynchronous events emanating in the host environment that effect the execution of a MAPSE process. Intrinsic in the design of the Event Monitor is the support for a multi-user environment consisting of processes that explicitly or implicitly depend upon execution stimulants external to the MAPSE. The interface to the Event Monitor is designed to be concise and portable, providing facilities that are adaptable to different host systems. Through the Event Monitor, the other functional domains of the KFW are designed to coordinate process execution so that service requests to the Kernel Process that cannot be performed immediately do not precipitate delays in the Kernel Process or the scheduling of another MAPSE process for execution.

The Event Monitor promotes an adaptive approach for handling manual process interruption that is compatible with the entry interrupt of the Ada language.

3.3.5.9 Volume Manager

The Volume Manager is designed to provide the KDBS with a concise portable interface for storing and retrieving data that comprises the object relationships and values in the data base. A straightforward abstract linear data structure is maintained through simple manipulative operations that are adaptable to mapping on a logical file management system or for implementation using the device handling subsystems of the host environment. The design of the Volume Manager relies on the facilities of the I/O Dispatcher to initiate the storage manipulation operations in the host environment. A consequence of this design is that the Volume Manager is isolated from issues related to concurrent data access by the KDBS at the logical object level and by the I/O Dispatcher at the host level.

Included in the Volume Manager is the editing of character transmissions to and from communication devices. The attention and break-in etiquette for process interruption is provided through the Event Monitor in a fashion conducive to its implementation on different host systems.

3.3.5.10 I/O Dispatcher

The I/O Dispatcher is designed to provide coordination of data transfer requests to the host environment from concurrently executing MAPSE processes. The interface supports requests that result from logical object manipulations in the KAPSE data base that have been reconciled to host system operations by the Volume Manager, or requests from a MAPSE process enclosing Ada low level I/C calls.

The design of the I/O Dispatcher is oriented to developing scheduling strategies for data transfer requests that promote their optimum handling by different host systems. In addition, an attendant requirement is to ensure that multiple requests performed for one process are not interleaved with requests from another when efficiency penalties would result because of access interference in the host system.

3.3.5.11 KAPSE Loader

The KAPSE Loader is designed to provide a flexible and adaptive approach for loading MAPSE processes into the host environment for execution. The Process Administrator starts a new process by instantiating an execution of the KAPSE Loader to load the new process from a host object that is in the Ada standard relocatable object format. This approach maintains the portability of the process load object and ensures that a new protected execution domain within the host system is created for a new process. A consequence of the design is that processes may be started through the MAPSE and execute solely in the host environment.

INTERIM TECHNICAL REPORT

TAB 2

KAPSE DATA BASE SYSTEM

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the KDBS preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements upon which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The KDBS is the cornerstone of the MAPSE and provides facilities for maintaining the different kinds of data needed in developing computer systems. The KDBS includes the facilities for storing the data as well as the data base management functions necessary to control data manipulation. All system and user-generated data is represented in the data base as objects and attributes. The KDBS is designed to ensure the correctness, consistency, and security of this data base.

The specific functional support areas within in the KDBS are object and attribute storage, partitions, categories, abstract objects and their versions, access control, Ada I/O, archiving, and backup.

1.2 DESIGN PRINCIPLES

Within the requirements to support the functional areas listed above, the fundamental goals of the KDBS design are to provide:

1. A standardized user interface to the data base
2. A data base structure that is portable and efficiently implementable
3. A secure, efficient access control mechanism
4. An integrated system for handling configurations, versions, and object histories

User interface standardization is of paramount importance. The portability of Ada programs between implementations of the MAPSE is possible only if the interface to the data base is rigidly standardized. This standardization requires that careful attention be paid to data base portability, and thus

any parameterization of the data base structure must be sufficiently innocuous to preserve the interface.

The data base structure must also be efficiently implementable, not only for a particular host, but for a range of hosts. There are two areas that deserve special attention with respect to storage efficiency: access control, and history maintenance. The principle that governed the design of access control was that the storage efficiency should be directly related to the degree of access control granularity. Thus, projects with uniform access control applied to broad partitions would incur minimum access control overhead. History maintenance, which is closely tied to version control, is an area subject to classical time/space tradeoffs. Because users with different needs would choose different points on the tradeoff curve, the design must provide flexibility in allowing users to explicitly identify the level of maintenance they require.

It has been advocated [1] that for the history attribute, all information about the construction for each object be kept. While this may be an obvious way to capture the information for object reconstruction, it requires disproportionate system overhead. A critical aspect of the KDBS is to design a method that supports object reconstruction without inundating the MAPSE with superfluous information. Collecting the command script, the referenced objects (including tools) and their versions should be sufficient to support the reconstruction of an object. A significant part of the design effort is to provide a system that substantiates this approach and permits a referenced object to be deleted safely.

SECTION 2 - BACKGROUND

The following section describes the background of this design effort. Previous work and literature have contributed to the design of the KDBS.

2.1 PREVIOUS WORK

The design of the KDBS extends the tradition of the UNIX™ and MULTICS file systems. In particular, the partition structure, access control mechanism, and version control owe a great deal to the corresponding UNIX concepts. In each case, however, these concepts have had to be modified and extended to satisfy the more comprehensive requirements of the MAPSE.

2.2 RELEVANT DOCUMENTS

1. Ada Support System Study (for the United Kingdom Ministry of Defence), Systems Designers Limited, Software Sciences Limited, 1979-1980.
2. Dolotta, T. A., S. B. Olsson, and A. G. Petruccelli, ed., UNIX User's Manual, Release 3.0, Bell Telephone Laboratories, June 1980.
3. Feiertag, R. J., and E. I. Organick, The Multics Input-Output System, Proc. Third Symposium on Operating Systems Principles, October 1971.
4. Fisher, David A., Design Issues for Ada Program Support Environments, Science Applications Inc., SAI-81-289-WA, October 1980.
5. Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.
6. Requirements for Ada Programming Support Environments - STONEMAN, United States Department of Defense, February 1980.
7. Revised Statements of Work for Ada Integrated Environment, Rome Air Development Center, 26 March 1980.
8. Ritchie, D. M., and K. Thompson, The UNIX Time-Sharing System, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.
9. Rochkind, M. J., The Source Code Control System, IEEE Transactions on Software Engineering, SE-1, December 1975.
10. Thompson, K., UNIX Implementation, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

SECTION 3 - FUNCTIONAL DESCRIPTION

This section provides a general overview of the KDBS interfaces within the MAPSE and describes the design tradeoffs performed.

3.1 INTRODUCTION

This paragraph discusses the system interfaces and functional design capabilities for this system element in terms of the design priciples outlined in Paragraph 1.2 of this tab.

3.2 SYSTEM INTERFACES

The KDBS has two levels of interfaces to the MAPSE system; those that are made available to MAPSE processes, and those required by the Kernel in order to process requests and remain transportable.

The KDBS facilities available to MAPSE processes are made visible through the KVI. These facilities are packaged into the Ada RTS Package and the KDBS Utility Package. The Ada RTS Package contains those functions necessary to support Ada Standard I/O as specified in the Ada Language Reference Manual. The KDBS Utility Package provides those functions necessary to manipulate and control the data base objects. These facilities are provided as separate packages because of their relationships to the MAPSE system. The KDBS strictly belongs to the MAPSE; it is designed for software development and has no place on a target machine. The Ada RTS Package is a portable package from the host MAPSE to the target machine, while those facilities found in the KDBS Utility Package are required only for the KDBS itself.

The above packages interface with the Kernel KDBS in order to provide the Kernel-level servicing required in order to process I/O requests. At the Kernel-level, the I/O servicing functionality is again partitioned, and there is an interface to the KFW. The KFW is responsible for mapping the logical KDBS and Ada I/O to physical I/O in the host environment. These interfaces are designed to maximize the portability of the KDBS portion of the MAPSE. All host-dependent functions are relegated to the KFW. Thus, aside from some simple parameterization, the KDBS and its interfaces are completely portable.

3.3 FUNCTIONAL CAPABILITIES

The KDBS provides the logical data base required by the MAPSE system. Its major functions support:

1. Partitions - To create, delete, and list partitions
2. Objects and attributes - To add, delete, and modify objects and attributes
3. Categories - To qualify names to denote the kind of object
4. Abstract objects and version control - To define and control abstract objects and their versions
5. Access control - To set and control the access control attributes of objects
6. Ada and KDBS I/O - To provide file and text I/O to support Ada Standard I/O and operations on KAPSE data base objects
7. Archives - To create and maintain archive objects
8. Backup - To backup and restore selected portions of the KDBS hierarchy.

Each of these functional areas is described in more detail in the following sections.

3.3.1 Partitions

A partition is a kind of KDB object analogous to a "directory" in many hierarchical file systems (see [6]). The partition object provides the mapping between the names of objects and the associated host files. A partition object is itself an object so that it can contain other partition objects (thus inducing a hierarchical data base structure). The MAPSE user creates subpartitions to contain groups of objects conveniently treated together.

The hierarchical structure of the KDB requires a partition from which all other objects can be reached. This partition is called the root of the hierarchy and is the starting point of searches for objects.

The unique object name is given by specifying the logical path from the root to the specified object. The root is specified by the "/" symbol, which is also used as the separator between other subpartitions. For example:

/RADAR/TRACK/PROG

names the object "PROG" in partition "TRACK", which is in turn a subpartition of "RADAR" under the root partition.

3.3.1.1 Current Working Partition

The KDBS permits references to objects to be relative to what is called the current working partition (CWP). The CWP is initialized for each MAPSE process, and the process can change it at any time. Within the ACLI, for example, the CWP may be changed by assignment to a command language variable:

%CWP := /RADAR/TRACK;

When a relative reference to an object is made:

SUB_SYS/TRAIN_PGM

the KDBS forms the absolute path to the specified object by linking the CWP to the relative reference to form the following absolute path name:

/RADAR/TRACK/SUB_SYS/TRAIN_PGM

3.3.1.2 Links

The major difference between the KDBS partition structure and the UNIX directory structure is the treatment of links. A link is a name in one partition that is used to access an object in another partition. The KDBS link is treated as a synonym for the absolute path name of the object, rather than as a direct link to the object itself. The present design is required to preserve security. With a direct link, it is impossible for the original object owner to prevent further access to an object from a user who has a link to it. The "alias link" requires each access to be subject to full access control.

3.3.2 Objects and Attributes

All data in the KDB is represented as objects: a KDB object corresponds exactly with an Ada file. All objects have attributes, which supply additional meta-information about an object. Much of this information is required by the KDBS to provide access control and configuration management, but user-defined attributes are also supported.

The KDBS supports logical object formats that provide both keyed and unkeyed access, and records of different lengths. This design contrasts sharply with that of UNIX, which provides only unstructured character stream files. While the UNIX design permits great flexibility in file handling at the user level, it does so at the expense of efficiency. Because the MAPSE is explicitly oriented toward programming-in-the-large (as opposed to UNIX's equally explicit orientation toward programming-in-the-small), efficiency assumes considerable importance. In fact, on many hosts it is anticipated that the efficiency of object accessing will be the most significant factor affecting the performance of the MAPSE. As a result, object formats that are conducive to efficient implementation must be provided. Therefore, the KDBS supports both keyed and unkeyed object access, and provides facilities for record I/O.

3.3.3 Categories

Categories provide a user-friendly means of dealing with object names. A category may be explicitly written as a qualifier to an object name:

prog'XQT or prog'TXT

However, most system tools will automatically supply a category for otherwise ambiguous names, based on context. For example, if the Compiler were asked to compile "prog", it would look for the source object "prog'TXT" and generate as output the relocatable object "prog'REL". There are twelve predefined categories for the MAPSE, and user-defined categories may be created as well.

3.3.4 Abstract Objects and Version Control

The version control system for the MAPSE is designed to manage changes made to textual and nontextual objects in the KDB. It provides facilities for storing, updating, and retrieving versions of objects, for controlling updating privileges, and for recording who made each change. The goals of version control are to provide for developer noninterference and multiple release maintenance, as well as to retain a program development history.

First, in programming-in-the-large, several developers will simultaneously need to access a given abstract object. For example, a developer may be testing one version of an object, while a second developer may be modifying a new version of the same object. Whenever possible, such access should be noninterfering.

A related requirement is that version control must permit multiple releases to be maintained. Typically, at some point in the development process a given version of an object will be designated a released version. Development may then proceed to generate a second release, but independent *maintenance of the first release is permitted.*

In order to permit reconstruction of previous versions of a program, and to maintain a development history, sequential versions of each module (object) must be maintained. These versions show the object's evolution. The last version in such a sequence is usually the preferred (or default) version, but it is possible to specify intermediate versions as well. Additional information can be associated with each version of a sequence, indicating who made the change and why it was made.

The above requirements for both sequential and parallel versions lead naturally to a tree-structured abstract object. Each branch of the tree can be considered to be a independently maintainable release. Each version along a branch represents an iteration of the corresponding release. Whenever necessary, a given version may be identified as a base for a new release, and a new branch can be sprouted at that point. Moreover, to provide for project control, the project administrator can restrict, on a

per-branch basis, permission to add new versions or to create new branches. These permissions are associated with the abstract object, and are independent of the usual access control mechanisms.

By definition, a version group is a set of objects that represent related iterations of a single abstract object. The name of the abstract object serves as a generic name for each version, and the abstract object itself serves as a directory for the version group. Version control is, however, an option to the MAPSE user. An object can be placed under version control when the object is created with the ACL command:

```
create(prog,delta);
```

The object "prog" is created in the CWP, and is placed under "delta" version control. This form of version control specifies the storage method to be used for the versions. Although this command is issued to the ACLI, subprograms are provided in the KAPSE Utility Package to perform the same functions. These subprograms are callable from MAPSE-level processes without involving the ACLI.

Object versions are named by qualifying the abstract object name with the branch and version names: object.branch.version. Incompletely qualified object version names are permitted, relieving users of the need to refer to specific versions. If only the branch name is specified, the last version on that branch will be taken as the default. If both the branch and version identifiers are omitted, the last version on the last-created branch is the default. This default scheme satisfies the most prevalent case, since most users will be accessing the last released version.

Version control is managed automatically by the KDBS whenever a request is made for an abstract object. As indicated above, the abstract object itself serves as a "directory" for all of its versions, and maintains the tree topology and creation permissions as well.

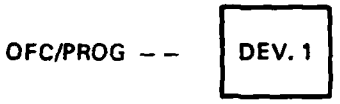
3.3.4.1 A Version Control Scenario

As an example of the use of version control, consider the development of a system by two teams of developers, a programmer team and a test team. The programmer team is responsible for developing and unit testing each module of the system. The test team is responsible for integration testing.

The program administrator creates an empty abstract object, "prog," in the "official" partition, "ofc". An initial development branch is created named "dev", and the programmer team is given permission to create versions on this branch:

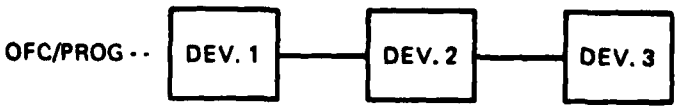
```
%CWP := "/ofc";              -- set CWP
create(prog'TXT,delta);      -- create the abstract object "prog"
                             -- with category "TXT"
createb(prog,dev);           -- create the branch "dev"
bwa(prog.dev,pteam,add); -- allows "pteam" to have branch
                             -- access (version create
                             -- permission) on branch "dev"
```

One programmer creates the first version of prog using the Editor. The version tree begins as:



During unit testing, a programmer discovers a bug and edits the dev.1 version of prog, creating version dev.2. Note that the programmer can simply edit ofc/prog, the dev.1 version will be selected by default, and the dev.2 version will be created by default. Also, when the Editor opens ofc/prog for editing, it actually reads from ofc/prog.dev.1 and opens ofc/prog.dev.2 for writing. Thus other programmers are temporarily prevented from adding a new version to the dev branch, and from interfering with development.

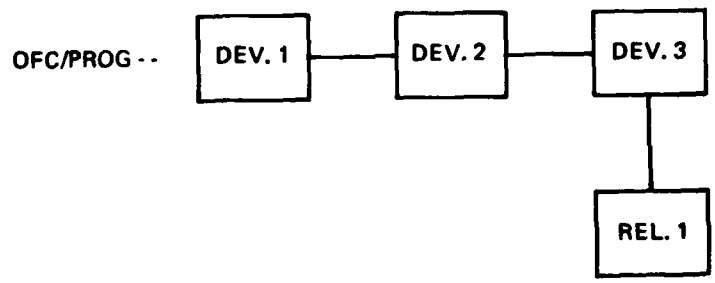
Version dev.3 may be created similarly, resulting in the tree:



At this point, dev.3 is successfully unit tested and turned over to the test team for integration. The test team cannot add versions to the dev branch, but a release branch called "rel" is created for them by the project administrator:

```
createb(prog.dev.3,rel);  -- create the branch rel,
                          -- stemming from version prog.dev.3
bwa(prog.rel,testteam,add); -- allow the testteam to add
                            -- versions to this branch
```

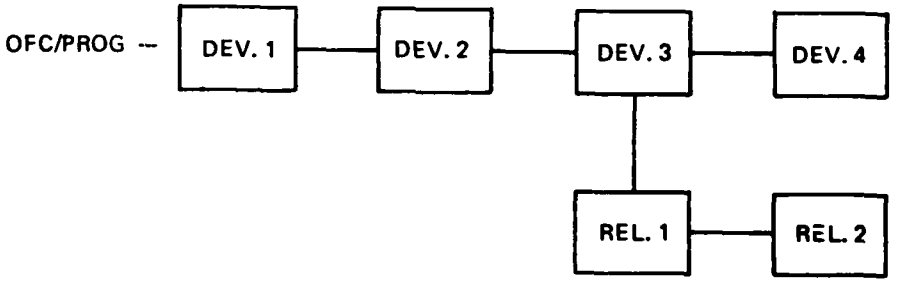
The initial version on the release branch, rel.1 is simply a placeholder, and equivalent to version dev.3. Note that the programmer team may not add versions to the rel branch. This ensures that the test team can always test from a stable base.



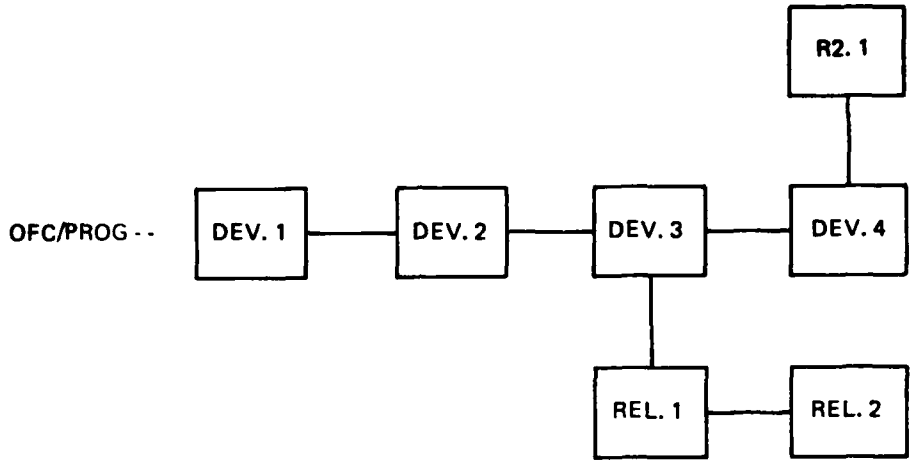
The test team integrates rel.1 and discovers a bug. Depending on the degree of project control desired, the project manager may then give the programmer team temporary permission to add a version to the rel branch. Alternatively, the programmer team may be required to perform their editing and unit testing in a separate partition. This latter method provides greater control; any changes must be unit tested to the satisfaction of the test team, who will then make the change to the rel branch themselves. A programmer can thus copy the current version on the rel branch to partition pgm, creating a new abstract object with the initial tree:

The programmer can edit this pgm/prog.rel.1 version, creating and successfully unit testing pgm/prog.rel.2:

When the test team is satisfied with the unit test, they can edit the official version /ofc/prog.rel.1, updating it with /pgm/prog.rel.2. Meanwhile, the programmer team may have independently continued to add a version to the dev branch. The official tree then looks like this:



The version control system has permitted the programmer team to continue incremental development, while allowing controlled maintenance of a release. Multiple releases present no additional problems, as arbitrary tree structures are permitted. For example, an "R2" release may be created based on dev.4:



3.3.4.2 Version Storage Options

For storing the versions, several options are available. First, each version may be kept as a separate object. This method provides for fast access at the expense of storage space. Second, the information content, but not the attributes, of a version may be deleted as long as it is defined in a configuration object and is thus reconstructible (configuration objects are discussed under the tab on the Configuration Management System in this report).

For text objects a third option is available. This is the delta method whereby only the differences between each version and its ancestor are recorded. These deltas are stored in the abstract object itself. The algorithm for delta maintenance ensures that only a single pass over the abstract object is required to extract any version. The advantage of deltas is that the differences between successive versions is usually very small, and thus the additional storage required for a version is correspondingly small.

The delta storage method is the default for text objects, and is unavailable for nontext objects. However, even with delta storage, frequently referenced object versions may be separately stored. Whether or not a text version is stored using deltas is invisible to the user, the KDBS automatically determines the necessary operations for version retrieval.

3.3.5 Access Control

The access control system used by the KDBS is a substantial extension to the system used by UNIX. The design provides a highly flexible, secure system. In addition, the design enables access attributes to be efficiently and compactly stored.

One of the first design issues was whether to associate access control information with users or with KDB objects. User-associated access control, aside from not satisfying the SOW requirements, was found to have several drawbacks despite its intuitive appeal. First, for a user to create an

object and restrict access to it, all user-associated access profiles must be changed. Permission to change these profiles is usually limited to system administrators or project managers. Thus it is difficult for users to effectively administrate their own objects. Second, objects may not be moved within the data base, or renamed, and still retain their access restrictions. Similarly, objects that are archived and subsequently retrieved into a different partition will find that their access restrictions have changed. The final objection to user-associated access control is that it is difficult, given any object, to determine which users have access to it. This determination involves searching the access profiles for all users, an extremely expensive operation.

Therefore, access attributes must be associated directly with the KDB objects themselves. In order to minimize the storage required for these access attributes, partition objects can be given partition access rights that apply to each object in the subtree rooted by the partition. If a finer degree of access control is desired, additional access rights can be individually assigned to subpartitions and member objects. Thus the storage required for access attribute storage is directly related to the granularity of control that is specified.

The access rights themselves include read, write, execute, append, modify, and delete. Append permission conveys the right to add to an object, and is distinguished from write permission. Modify permission allows modification of the object's attributes. Sets of these access rights may be specified for individual users, user groups, or all other users.

Because partitions have rights that are distributed to an object's members, the access permissions for objects are determined when it is accessed. The rights for the particular form of access are extracted from each containing partition and intersected. With this algorithm, detailed in the KDBS B5 Specification, access to an object can never be less restrictive than that specified by its containing partitions. This rule gives project management the ability to selectively delegate access rights, while permitting users to administer and control their own objects within the delegated restrictions. Users can therefore exercise discrete control only within those portions of the KDB hierarchy where they themselves have the proper access rights.

When access to an object is requested by a MAPSE process, the access permissions associated with the object are determined for the effective user and user group identifiers associated with the process. The ACLI is initiated with its effective identifiers set to identify the logged-in user. When a process is initiated, the effective identifiers of the invoked process are ordinarily copied from those of the invoking process. The invoked process will thus have the same access rights as the invoker. It is possible, however, for a user to create an executable load object and assign it the "set effective ids" attribute. This attribute will ensure that when the object is executed as a process, it will have its effective identifiers set to the owner and group identifiers of the object. For example, a user may own a data object that others may access only by executing a program also belonging to the same owner. Thus, with the "set effective ids" attribute, the rights accorded to a process are those of the owner of the associated load object, not of the user who invoked the process.

3.3.6 Ada Input/Output

Input/Output facilities are predefined in the Ada language in two packages. The generic package INPUT_OUTPUT defines a set of input/output primitives applicable to files containing elements of a single type. Additional primitives for text I/O are supplied in the package TEXT_IO. These packages are supplemented with text formatting packages that are described in detail in the B5 Specification for the Compiler.

The context of execution for Ada I/O, supported by the KDBS, is divided between two domains: the high-level subprograms that comprise the Ada RTS Package and KDBS Utility Package are called from executing Ada programs and execute at the MAPSE process level; the underlying support routines execute at the kernel level. There are several reasons for this separation. First, I/O within the MAPSE is interrupt-driven and the requesting MAPSE process may not always be in main memory when the requested I/O has been completed. In this way, other MAPSE processes (or tasks) can be scheduled and executing while other MAPSE processes wait for their I/O requests to complete.

Moreover, it is usually not feasible to allow user processes to field their own interrupts. Another reason is that the high-level packages are all designed to be highly portable, because they must be provided on target systems as well as in the MAPSE. The interface between these packages and the underlying KDBS I/O facilities is specified in detail; I/O facilities that support this interface must be provided for each target. Finally, the KDBS I/O support is required to handle version storage, while this is unnecessary for target systems. The processing required to read a version under delta storage is provided entirely within the kernel.

The KDBS facilities defined in the Ada RTS Package to do I/O are designed to eliminate the differences between the various devices and styles of access found on each implementation of the MAPSE. An object exists in the the KDB for each type of device supported, so that the structure of a device name is the same as that of any other object name. Moreover, the same access protection afforded to objects of the KDB apply to the device objects as well.

Names exist for terminal devices, and may exist for disk devices or tape drives regarded as physical units outside of the KDB. Since terminals and other objects are treated identically, both may use the same I/O calls. Thus it is easy to redirect input and output from the terminal to another object. Moreover, it is equally easy to implement "pipes", which redirect output from one process to be input to another process.

The KDBS maintains no locks visible to the user, nor is there any restriction on the number of users that may have a specific object open. There are, however, sufficient internal locks to maintain the logical consistency of the KDB when two users try to update the same object, create objects of the same name in the the same partition, or delete objects that are currently being used.

3.3.7 Archive

The archive facility is provided to allow users to selectively save KDB objects on backing storage. Archiving is particularly useful as a means of reducing online storage requirements for infrequently accessed objects. Archiving is also convenient for packaging sets of related objects on removable media for transporting between installations.

The archiving facility provides a full complement of functions that operate on archive objects to list the members, to add members, to replace members, to update members only with a more recent version, to delete members, and to retrieve members.

In order to preserve data base security, users may request archiving only for those objects to which they have full access.

3.3.8 Backup

The backup facility is provided to minimize loss of data due to hardware failures and inadvertant deletion or modification of objects by users. The backup function dumps the entire KDB hierarchy or selected portions of the hierarchy onto a removable medium. Backup thus establishes a checkpointed data base, which is formatted to permit selective restoration of lost or damaged objects. Backup may be activated automatically by a timer-activated process, or may be called by the user. In any case, the backup operation must wait until the desired portion of the data base is inactive, and must lock out any further activity until the backup is complete. User cooperation may be required to achieve data base inactivity, or this may be forcibly accomplished by the system administrator.

As with the archive facility, users may request backup only for those objects to which they have full access.

INTERIM TECHNICAL REPORT

TAB 3

APSE COMMAND LANGUAGE INTERPRETER (ACLI)

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overvie of the ACLI preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements on which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The ACLI is the most visible communication channel between the MAPSE and its users. The ACLI interprets the ACL, which is a command programming language providing a flexible user interface to the process and data base management facilities of the KAPSE. ACL is specifically designed to be usable by both project management and software development personnel.

The ACLI is designed to provide an efficient, powerful, user-friendly mechanism for invoking MAPSE programs. The ACLI interprets ACL, which combines both programming language and command language constructs. The programming constructs are those usually found in most algorithmic languages, and include variables, aggregates, assignments, control-flow primitives, and subprograms with parameter passing. The command constructs provide access to the process management and data base management facilities of the KAPSE.

Many of the ACL commands are directives to the ACLI to establish a named program as a MAPSE process. These commands permit the user to initialize the environment in which processes are invoked, and to redirect their standard input, standard output, and standard error files. A few commands are directly executed by the ACLI to alter the state of the ACLI process. In addition, the ACLI may, for efficiency, execute directly some frequently used commands that could also have been implemented as separate programs.

The ACLI executes as an ordinary MAPSE process, and possesses no special privileges. Thus the ACLI is not part of the KAPSE, and users are free to substitute their own MAPSE programs to execute in place of the ACLI.

In order to provide a user-friendly environment for debugging, the Debugger is provided as an integral part of the ACLI. Since the Debugger contains considerable functionality in its own right, it is treated under a separate tab in this Report.

1.2 DESIGN PRINCIPLES

The overriding requirements influencing the design were, of course, presented in the Statement of Work (see Paragraph 3.2.8.2 in [12]) and STONEMAN (see Paragraphs 4.C, 4.D, and 5.D in [11]). Within these constraints, the design was directed toward providing a command language with the following characteristics:

1. User-friendly syntax for frequently executed contructs
2. Organized around simple and straightforward concepts that are few in number
3. Easily usable by project management personnel who may not be Ada programmers
4. Sufficiently powerful for implementing sophisticated command language subprograms.

Given the need for having project management and program support personnel online, it was recognized that the ACLI must be usable by people not versed in Ada programming. The resulting simplicity need not, however, be pervasive. ACL is carefully organized toward providing simple-to-use facilities for casual users, while giving sophisticated users many of the constructs found in algorithmic languages.

The suggestion has been made that the command language should be an interpretable version of Ada [5]. Aside from violating the principle enunciated in the previous paragraph, an Ada command language would suffer from other problems. These problems all result from a divergence in the design goals of programming languages and command languages. Ada was designed to be a compilable, strongly type-checked, algorithmic language for embedded systems. Command languages, on the other hand, are used as vehicles for program development, for initiating, monitoring, and debugging

processes. Command languages are thus highly interactive by definition, Ada is not. Most importantly, a user-friendly mechanism for invoking and composing the invocations of processes is intrinsic to command languages, this facility is unavailable in Ada.

Nevertheless, in order to provide sufficient expressive power to combine and monitor process invocations in nontrivial ways, a number of constructs common to programming languages must be included in the command language. Wherever possible, these constructs were adapted, if not borrowed, from Ada. A substantial effort was made to ensure that any differences would not be likely to lead to user confusion and error.

Deviations from Ada syntax were never be made purely for reasons of stylistic preference. Deviations were only permitted in those areas where the candidate Ada constructs clearly did not satisfy the fundamental needs of a command language.

Since the ACLI is a MAPSE program, it has access through the KVI to the facilities of the KAPSE. It is important to note that the user has an alternative access to the KAPSE by writing, compiling, and executing Ada programs. However, MAPSE users need not all be Ada programmers. The ACLI must therefore cater to the needs of managers and program support personnel as well.

One aspect of process initiation for which the ACLI is responsible is that of environment initialization. Part of the process state for each process is a description of how that process is connected to the system. This description includes data describing terminal characteristics, standard I/O devices or objects, interrupt vectoring, parent and child process identification, etc. This information is initially set by the ACLI, and is available to the process through an Ada Standard Environment package definition.

SECTION 2 - BACKGROUND

The following section provides a description of the background of this design effort. Previous work and literature have contributed to the design of the ACLI.

2.1 PREVIOUS WORK

The fundamental concepts present in both the ACLI and ACL are derived from the highly acclaimed UNIX™ Shell and its command language. In fact, a major goal of this design effort was to achieve an elegant blend of Shell-like concepts with Ada-like syntax. In addition, the early decision to treat the ACLI as a MAPSE program and not as part of the KAPSE was based entirely on the success that this concept has enjoyed in UNIX.

UNIX has had a number of Shells, and the features of the original Shell, the PWB-UNIX Shell, the "Bourne" Shell, and the Berkeley C Shell were all examined. A lesser, and sometimes indirect, debt is owed to the Cambridge Multiple Access System, CTSS, and MULTICS.

2.2 RELEVANT DOCUMENTS

The following documents are all related to the design and the design history of the ACLI and ACL:

1. Ada Support System Study (for the United Kingdom Ministry of Defence), Systems Designers Limited, Software Sciences Limited, 1979-1980.
2. Bourne, S. R., The UNIX Shell, The Bell System Technical Journal, Vol. 57, No.6 Part 2, July-August 1978.
3. Crisman, P. A. (ed.), The Compatible Time-Sharing System, M.I.T. Press, 1965.
4. Dolotta, T. A., S. B. Olsson, and A. G. Petruccelli, UNIX User's Manual, Release 3.0, Bell Telephone Laboratories, June 1980.
5. Fisher, David A., Design Issues for Ada Program Support Environments, Science Applications Inc., SAI-81-289-WA, October 1980.

6. Hartley, D. F. (ed.), The Cambridge Multiple Access System — Users Reference Manual, Cambridge University Mathematical Laboratory, 1968.

7. Joy, W., An Introduction to the C Shell, Dept. of Elec. Engr. and Computer Science, Univ. of California, Berkeley, 1980.

8. Mashey, R., Using a Command Language as a High-Level Programming Language, Proc. 2nd Int. Conf. on Software Engineering, October 1976.

9. Organick, E. I., The MULTICS System, M.I.T. Press, 1972.

10. Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.

11. Requirements for Ada Programming Support Environments - STONEMAN, Department of Defense, February 1980.

12. Revised Statements of Work for Ada Integrated Environment, Rome Air Development Center, 26 March 1980.

13. Ritchie, D. M., and K. Thompson, The UNIX Time-Sharing System, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

14. Wegner, P., The Ada Language and Environment, Software Engineering Notes, Vol. 5, No. 2, April 1980.

SECTION 3 - FUNCTIONAL DESCRIPTION

This section provides a general overview of the ACLI interfaces within the MAPSE and describes the design tradeoffs performed during the ACLI design.

3.1 INTRODUCTION

This paragraph discusses the system interfaces and functional design capabilities for this system element in terms of the design priciples outlined in Paragraph 1.2 of this tab.

3.2 SYSTEM INTERFACES

The ACLI uses the facilities of the Ada Standard I/O Package, the KDBS Utility Package, and the KFW Interface Package. The ACLI is initiated by the KFW Logon Process, and may also be invoked by any MAPSE-level process.

3.3 FUNCTIONAL CAPABILITIES

The ACLI serves as the user interface to the facilities of the MAPSE - to the process initiation and control functions provided by the KFW, and to the data base management functions provided by the KDBS. The ACLI possesses the same potential access rights to these functions as any other MAPSE tool; naturally, certain kinds of access may be restricted to particular users or programs. The ACLI executes as a MAPSE process, and may initiate other MAPSE processes. The ACLI provides the user the option of either suspending the ACLI instance pending the completion of another process, or continuing to execute asynchronously.

It must be noted that process initiation by the ACLI is not the same as Ada task activation. A process is an initiated Ada program, and by definition an Ada program cannot be a task. The logical control of Ada tasks is solely the responsibility of the program in which those tasks are declared. The Ada language provides mechanisms for task management, and these mechanisms use primitives that are part of the RTS Package.

The Debugger is an integral part of the ACLI, and enables operational programs to be debugged. That is, the Debugger need not be called before the invocation of the program to be debugged. As indicated above, the

functionality of the Debugger is discussed under a separate tab in this report.

There are a number of command utilities that are provided to complement the functionality of the ACLI. These constitute an initial, minimal set of programs that make available to the ACLI user the requisite functionality of the KFW process control facilities and the KDBS data base management facilities. The utilities are all invoked as process calls. Most will execute as separate processes; however, a few (the Debug call, for example) affect the execution state of the ACLI and will be executed directly within the ACLI.

There are two major functional capabilities to be discussed. Paragraph 3.3.1 will present the design of the ACLI and of its role in the MAPSE. Paragraph 3.3.2 will concentrate on the design of ACL itself.

3.3.1 ACLI Design

As has been emphasized above, the ACLI is simply another tool in the MAPSE tool set, and is not part of the KAPSE operating system. The ACLI thus enjoys no special privileges, and has the same access as any other MAPSE program to the KFW and KDBS facilities provided through the KAPSE virtual interface. This arrangement has two major advantages. While the Kernel must be resident, the ACLI need not be. Thus the size of the ACLI is not particularly important. Moreover, other MAPSE programs are not required to invoke the ACLI in order to request Kernel services.

The ACLI is usually invoked by the Logon Utility of the KFW. After verifying the user's identity through a password protocol, the Logon Utility invokes the program named in the user's password entry record. This program is ordinarily the ACLI, but the user may specify a different program if so desired. Users may thus define their own command languages, and new or nonstandard versions of the ACLI may be tested without impairing the reliability and security of the KAPSE. In addition, limited versions of the ACLI may be provided for users who are to be restricted from accessing the full functionality of the KAPSE.

The user may provide a profile object to be processed by the ACLI immediately after logon. The commands in this object will typically initialize environment variables to tailor the invocation of the ACLI to the user's needs.

3.3.2 ACL Design

ACL is both a command language and a programming language. The command processing facilities are, however, the reason for the existence of ACL. As will be seen, the command constructs are responsible for most of the differences between ACL and Ada. The discussion of the design of ACL parallels the order of presentation in the ACL Reference Manual.

3.3.2.1 Lexical Elements

ACL commands are formed by combining data base object names, constants, variables, and operators. Names, constants, and variables must be distinguished from each other: names are unadorned, string constants may be enclosed in quotes, and variables must be prefixed with a "%". Context will always allow distinction between names and unquoted constants. Strings need be quoted only to prevent interpretation of their contents as ACL commands.

The above convention is a deviation from Ada, which quotes object names and leaves variables unadorned. In command languages, however, object names are used far more frequently than variables, and so deserve the unencumbered syntax. This convention is unlikely to cause confusion, because the use of quotes is superfluous, not erroneous.

A number of identifiers are reserved, and the Ada preference for full English words over abbreviations is followed. Special characters, blanks, horizontal tabs, and line breaks all serve as delimiters. ACL commands are entirely free-format, and statements are terminated with a semicolon. Although a user may thus divide a statement across several lines, no portion of the statement is interpreted by the ACLI until the entire statement is read.

Object names have a somewhat specialized syntax. The simplest object name is an identifier:

prog

prog names an object in the current partition, the default partition in which names are looked up. In general, an object name consists of sequence of identifiers where all but the last identifier names a partition. Thus:

/radar/tracking/prog

names the object "prog" in partition "tracking", which is a subpartition of "radar", which is a subpartition of the root "/". The partition structure is discussed in more detail under the KDBS tab of this interim report.

Names that identify abstract objects (objects with versions) may be qualified to specify particular versions. For example:

prog.ibm370.5

denotes the fifth version on the branch "ibm370" of the abstract object "prog".

prog.ibm370

denotes the last version on branch "ibm370", and

prog

denotes the last version on the default branch of the abstract object "prog". A full discussion of objects and versions may also be found under the KDBS tab of this report.

Simple patterns may be used to generate lists of object names. The "*" will match any sequence of characters in a component of a name. For example, if the current directory contained prog1, prog2, and progabc:

prog*

would generate the list (prog1 prog2 progabc) as an aggregate. Similarly:

/radar/*/prog.ibm370.*

would generate a list of all versions on the ibm370 branch of each abstract object prog appearing in any subpartition of the partition radar.

Two special object names are provided. "." identifies the current partition, and ".." identifies the partition containing the current partition.

3.3.2.2 Variables, Types, and Expressions

ACL is a typeless language, and there are no declarations. This is a radical departure from Ada, necessitated by the requirement that ACL be usable by the non-Ada-programmer, and friendly to the Ada programmer. Thus a project manager can login and type:

printreport(myproject);

without declaring either the program "printreport" or the partition "myproject". Virtually all of the interaction of the nonprogrammer with the ACLI will be to invoke canned procedures using a similar syntax.

Variables are declared when they appear on the left of an assignment, or in an out parameter position in a process call. Only two types may occur as the values of variables: strings and aggregates of strings. This lack of a complex typing structure simplifies ACL considerably without affecting its usefulness as a command language. Admittedly, the programming aspects of ACL suffer, but the ACLI is not the only pathway to the functionality of the KAPSE. The user who requires programming capabilities beyond the scope of ACL can always write an Ada program that interacts directly with the KAPSE.

Strings are handled as in Ada, and the string catenation operator, "&", is available. There are, however, some differences. Strings need be quoted only if they contain delimiters significant to ACL, and variables do not have a predeclared maximum string length (although a host-dependent parameter may limit string length).

Aggregates may be delimited by blanks or commas:

(abc,def,ghi) or (abc def ghi)

Simple Ada-like slices are also available:

```
%a := (abc def ghi);
%b := %a(2..3);
     -- the value of %b is now (def ghi)
```

The syntax for expressions is borrowed, almost in its entirety, from Ada. Thus Boolean operators (and, or, etc.), membership operators (in, and not in), relational operators (<, >, etc), arithmetic operators, and the string catenation operator are allowed.

There is, of course, no overloading of operators. Each operator will convert its operands, if necessary and possible, to conform to the required operation. Thus, for example, the string "15" may appear as an operand to the string catenation operator as well as to an adding operator. Boolean operators interpret their operands as Boolean values. The only operands interpreted as FALSE are the string "0" and the aggregate with the string "0" as its only component. All other operands are interpreted as TRUE. Boolean, relational, and membership operators always generate a result of "1" for TRUE and "0" for FALSE.

3.3.2.3 Process Invocation

A process call is the mechanism that causes the execution of an Ada program as a MAPSE process. The simplest form of a process call specifies the names of the program and its arguments:

```
list(/radar/tracking);
```

This call would list on the Standard Output file the names of the objects in partition /radar/tracking. Note that compatibility requires that the parameter passing mechanism used by ACL conform to that used by Ada. Ada allows parameters to be passed to a main program that is a function or a procedure. Although parameter passing is an extra burden on the underlying KFW process invocation facility, this use of parameters provides a feature of considerable power.

ACL provides a means to redirect the standard input and standard output files. One form of redirection specifies that the output of one process be "piped" to the input of a second process. For example:

```
list(/radar/tracking)  | sort();
```

This pipeline would sort the object list created by "list" before printing it on the terminal. Process calls separated by "|" characters result in

separate processes being created. The standard output of the first enters a pipe, which is a special temporary buffer created by the KAPSE. The standard input of the second process is read from the pipe, and synchronization of the processes is handled automatically by the KAPSE. Thus, for the above example, "list" is suspended when the pipe is full, and "sort" is suspended when the pipe is empty. Many process calls may be connected in a pipeline, although the KAPSE may limit the number of processes a single user may activate.

The standard input and output files may be redirected to other data base objects (or devices) as well. For example:

```
list(/radar/tracking) -> list1;
list1 -> sort() -> list2;
```

Here, the standard output of "list" will be written into the data base object "list1". For "sort", the standard input will be read from "list1" and the standard output will be written into "list2".

Given this pipeline facility, it is desirable that error and warning messages should appear on a file other than standard output. The file standard error is provided for this purpose, and is ordinarily connected to the user's terminal. Thus, in the "list sort" pipeline above, the user would see any warnings from "list" on the terminal; they would not be sent down the pipeline for "sort". If desired, the standard error file may be redirected to a data base object for each process call in a pipeline:

```
list(/radar/tracking) *> warnings  | sort() ;
```

Here, the object named "warnings" would receive all error or warning messages from the "list" process.

Redirection of standard output or standard error may also specify appending to an existing object with the operators "->>" and "*>>", respectively.

For process calls, in, out, and in out parameters are all permitted. The form of the parameter is determined by the specification in the called program. The ACLI arranges all parameter transmission with the aid of KAPSE facilities. In addition, a process call that names a function may return a

value. By convention, for system tools, this return value will indicate whether the process executed successfully or terminated abnormally.

Ordinarily, the ACLI will invoke a process (or pipeline) and await completion. The user may, however, specify that a process is to be executed asynchronously, in which case the ACLI will not wait for the process to finish. Asynchronous, or background execution is accomplished by enclosing the process call or pipeline in parentheses:

```
( list(/radar/tracking) | sort() -> list );
```

To avoid confusion, it is generally a good idea to redirect the output of asynchronous programs away from the terminal.

It is possible to substitute the standard output of a process back into the command stream as a string or as an aggregate. For example, if the "sort" program sorted an aggregate provided as an argument instead of its standard input, the following would be equivalent to the original example:

```
sort(# list(/radar/tracking) #);
```

The ACLI would reformat into an aggregate the Standard Output of the "list" process call. In the reformatting, blanks characters, horizontal tabs, and newlines serve only to delimit the strings that become the components of the aggregate. The syntax " '# . . . #' " is provided to preserve the standard output as a string and include the above delimiters.

3.3.2.4 Statements

Although process calls and pipelines are the most common forms of statements, ACL provides a full complement of programming constructs. It should be noted that most of these constructs are designed to be used in ACL subprogram objects. While they may be entered and interpreted "on the fly", experience with the UNIX Shell indicates that the large majority of users direct requests are for process invocations, process pipelines, and I/O redirection.

Simple statements include the assignment statement, which assigns a string or aggregate to a variable. The following are all valid assignment statements:

```
%a := abc;
%b := 25;
%c := 14 + %b;        -- c now has the value "39"
%d := %a & %c;        -- %d now has the value "abc39"
%e := (%a,%d);        -- %e now has the value (abc,abc39)
```

Since process calls may return a value, these values are assignable:

```
%retval := list(/radar/tracking);
```

In this case the return value, by convention, might encode "0" for successful termination, "1" for "cannot access partition", "2" for "partition does not exist", etc. (note that these error messages may be written to standard error as well).

Assignment defines the variable on the left side, and reassignment constitutes a new definition. A variable with a string value may thus be reassigned an aggregate value.

A number of control-flow statements are provided, as noted above they are provided primarily for use in ACL subprograms (see *Paragraph* 3.3.2.5 below). There is the Ada-like conditional statement:

```
if expression then
    sequence-of-statments
  elsif expression then
    sequence-of-statements
  else
    sequence-of-statements
end if ;
```

Any number (including zero) of "elsif" clauses may be specified, and the "else" clause is optional. As in Ada, the "if -- end if" bracketing obviates the dangling-else ambiguity.

Case statements are particularly useful for testing values returned from process calls. If, for example, "list" did not write its error messages, the following would test the value returned and take appropriate action:

```
case list(/radar/tracking) -> list is
     when "0" => null;
     when "1" => echo("cannot access file");
     when "2" => echo("file does not exist");
end case ; :
```

The standard output of "list" would be written to the object "list", but the success of its execution would be monitored by the case statement. The "null" statement is provided expressly for these situations, and the "echo" program writes its argument to its standard output.

Several forms of loop statements are available. Simplest is an endless loop that can be exited only via "exit" or "return" statements:

```
loop
     sequence-of-statements
end loop;
```

"While" loops are permitted:

```
while expression
     loop
          sequence-of-statements
     end loop;
```

A form of "for" loop is provided that allows iteration through components of an aggregate. For example:

```
for %i in # list(/radar/tracking) #
     loop
          if Ada(%i) /= ok then
               echo("error in compiling object & " %i);
               exit;
          end if;
     end loop;
```

This statement would compile each object in the /radar/tracking partition, until a compilation resulted in an error.

A loop may specify a loop identifier as in Ada, and an exit statement is provided:

exit loop-identifier when expression;

Both the loop identifer and the expression are independently optional. The loop identified with the loop identifier is exited, unless the loop identifer is absent, in which case the innermost enclosing loop is exited. If the when clause is specified, loop termination occurs if and only if the expression is TRUE.

3.3.2.5 Command Language Subprograms

Command language subprograms may be specified in ACL. An ACL subprogram is invoked with the syntax of a process call, and the ACLI checks the category of the named program object to determine whether the call is for an Ada program or for an ACL subprogram. To invoke an ACL subprogram, the ACLI creates another ACLI process to interpret the named object.

For example, the following is a subprogram, copar, that compiles and links all objects within a partition. It can be invoked by "copar(partition name)".

```
function(%partition : in) is
     for %i in # list(%partition) #
          loop
               if ada(%i) /= ok then
                    return "bad compile" & %i;
          end loop;
     %s := link(# list(%partition) #, PAROUT);)
     if %s = ok then
          return success;
     else
          return "link failure";
```

Invocation would then be as in a normal process call:

copar(/radar/tracking);

Standard input and standard output behave as for normal process calls, so ACL subprograms may appear in pipelines. Subprograms specified with the function keyword, rather than the procedure used above, may return values:

return expression;

As indicated above, ACL subprograms are invoked by creating another instance of the ACLI process. Such an invocation does not permit the subprogram to directly access the standard environment of the calling process. Sometimes, however, it is useful to execute a subprogram inline. Such an invocation does not create a new process, but instantiates the text from the named object with parameters handled appropriately. The inline statement has the following syntax:

. mcl-program-name (parameters);

Execution is in the environment of the current instance of the ACLI. Inline calls may not appear in pipelines because of the clash with the semantics of standard input and standard output.

INTERIM TECHNICAL REPORT

TAB 4

CONFIGURATION MANAGEMENT SYSTEM

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the CMS preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements upon which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The CMS is responsible for processing and maintaining configurations and object histories. The system consists of a MAPSE-level tool, Configure, which utilizes underlying KDBS functions to manage the history attributes.

The CMS automates many of the configuring activities involved in project development, testing, and maintenance. This automation is accomplished through a careful integration of the notions of configurations, versions, and history attributes.

A configuration is a set of data base objects combined with a set of rules that specify how these objects may be derived from each other. A configuration is defined by a configuration object, which is prepared by the user. The Configure tool uses the configuration object to provide two major *functions:*

1. Update - Determine and perform the minimal set of operations required to bring specified objects up to date with the other objects in the configuration.

2. Reconstruct - Determine and perform the minimal set of operations that are required to reconstruct a specified version of an object in a configuration.

Configure calls on the ACLI to execute the indicated operations, which are written in ACL.

Central to the operation of Configure is the maintenance of the history attributes. These objects record how an object was constructed, and permit its reconstruction. The history attributes also prevent the removal of an object necessary to the reconstructability of other objects.

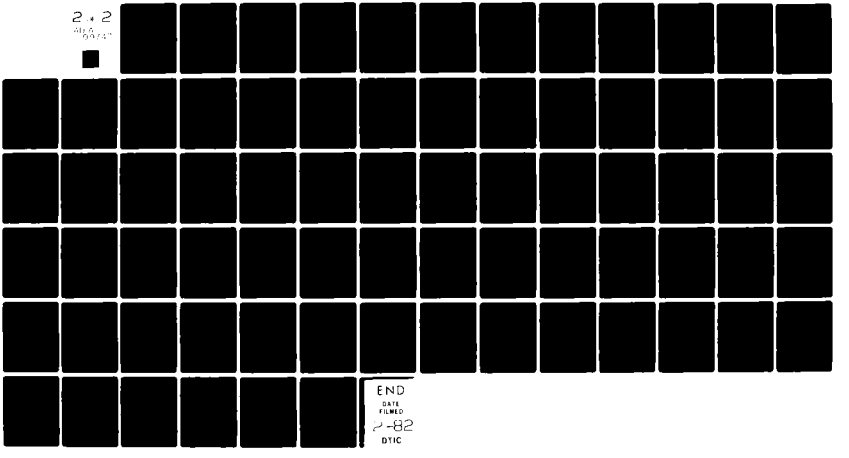
The next paragraph will describe the design principles that provided the basis for the design of the CMS. Section 2 will document previous related work. Section 3 will present the rationale that guided the design decisions, and give examples illustrating the use of Configure.

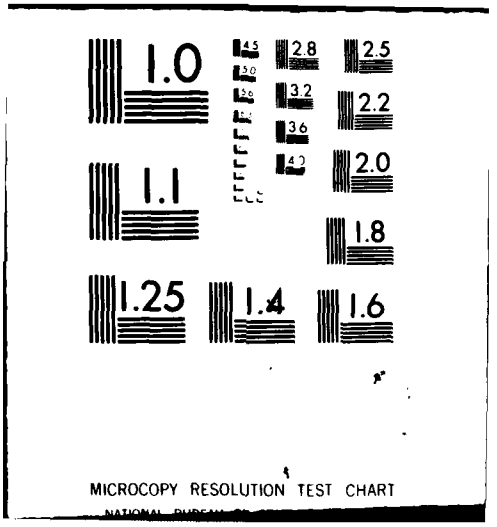
1.2 DESIGN PRINCIPLES

Large projects typically require the development of many programs. These programs, in turn, are commonly divided into a number of smaller modules. The construction of a given program from its constituent modules often entails a lengthy sequence of compilation and linkage steps. This sequence may be further complicated by special options or by the use of program generators such as parser or scanner generators. Many dependencies are embedded in the construction sequence, and the order in which the steps are performed is critical. Moreover, after a program has been constructed, the testing and maintenance cycles will modify the constituent modules. It is difficult for a developer to recall exactly which modules will have to be rebuilt, and errors in this process tend to generate bugs that are particularly difficult to locate. Recompiling everything is expensive, and often not practical for large projects.

The requirements regarding configuration management in both the SOW and STONEMAN left open many design issues. The present design is specifically oriented toward solving the problems of project and configuration management presented in the above paragraph. The CMS is designed to achieve the following goals:

AD-A109 747 COMPUTER SCIENCES CORP FALLS CHURCH VA F/G 9/2
ADA INTEGRATED ENVIRONMENT II.(U)
DEC 81 F30602-80-C-0292
UNCLASSIFIED RADC-TR-81-363 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART

1. Configurations should specify how member objects are derived from each other

2. Configurations should interact closely and naturally with the concept of object versions

3. Configurations should interact closely and naturally with the concept of object history attributes

4. The interaction between configuration management and history attributes should enable the minimization of the storage required for history preservation

5. Configuration management should integrate the concepts of object generation and object reconstruction

6. Configuration management should be useful in every stage of project development: initial program development, unit testing, integration testing, and maintenance.

In achieving each of these goals, simplicity and efficiency were the primary concerns. For example, an early design effort [1] suggested that lengthy derivation scripts be stored with each object in order to preserve generation history. While this approach is theoretically sufficient, it imposes unrealistic storage requirements for large projects. In short, the aim of the present effort was to achieve the above design goals within the confines of a practical and implementable system.

SECTION 2 - BACKGROUND

The following section provides a description of the background of this design effort. Previous work in this area as well as literature have contributed to the design of the CMS.

2.1 PREVIOUS WORK

The fundamental concept for Configure was derived from the Make facility of UNIX™. The UNIX Make permits the specification of how files are to be derived from other files. The present design has successfully solved the problem of adapting the Make concept to handle object versions, manage history attributes, and provide for object reconstruction. In general, this adaptation was an extension. However, Make was not designed to handle object reconstruction, and therefore permits considerable freedom in derivation specifications, which the present design has restricted. In particular, Configure does not permit user interaction during the derivation of an object defined in a configuration. This restriction guarantees reconstructability using the derivation script maintained in the configuration object.

2.2 RELEVANT DOCUMENTS

1. Ada SUPPORT SYSTEM STUDY (for the United Kingdom Ministry of Defence), Systems Designers Limited, Software Sciences Limited, 1979-1980.
2. Dolotta, T. A., S. B. Olsson, and A. G. Petruccelli, ed., UNIX USER's Manual, Release 3.0, Bell Telephone Laboratories, June 1980.
3. Feldman, S. I., Make -- A Program for Maintaining Computer Programs, in Documents for the PWB/UNIX Time-Sharing System, Edition 1.1, Bell Telphone Laboratories, October 1977.
4. Fisher, David A., Design Issues for Ada Program Support Environments, Science Applications Inc., SAI-81-289-WA, October 1980.
5. Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.
6. Requirements for Ada Programming Support Environments - STONEMAN, United States Department of Defense, February 1980.
7. Revised Statements of Work for Ada Integrated Environment, Rome Air Development Center, 26 March 1980.

8. Ritchie, D. M., and K. Thompson, The UNIX Time-Sharing System, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

9. Rochkind, M. J., The Source Code Control System, IEEE Transactions on Software Engineering, SE-1, December 1975.

SECTION 3 - FUNCTIONAL DESCRIPTION

The following section provides a general overview of the CMS interfaces within the MAPSE and describes the design tradeoffs performed during the Configuration Management System preliminary design.

3.1 INTRODUCTION

This paragraph dicusses the system interfaces and functional design capabilities for this system element in terms of the design principles outlined in Paragraph 1.2 of this tab.

3.2 SYSTEM INTERFACES

The CMS interfaces with the ACLI and the KDBS. The ACLI interprets the Configure commands. The KDBS is the repository of all the data, attributes, etc., that Configure uses in its configuration management function.

3.3 FUNCTIONAL DESCRIPTION

The functionality of the CMS depends on the integration of configurations with object versions and history attributes. Paragraph 3.3.1 discusses this integration, along with the role of the ACLI, in the context of a simple example. An advanced example is provided in Paragraph 3.3.2 to further illustrate the power and versatility of Configure.

3.3.1 Configurations, Versions, and Histories

Consider the following configuration object (CO), which describes how an executable object is derived from a relocatable object, which in turn is derived from a text object:

```
prog'XQT  :  prog'REL
     link(prog,NAME=>prog);
prog'REL  :  prog'TXT
     Ada(prog);
```

The left-justified lines specify dependencies, and the indented command lines define the operations, written in ACL, required to fulfill the dependencies. This CO indicates that the relocatable program prog'REL

depends on prog'TXT, and is built by compiling prog'TXT. Similarly, prog'XQT depends on and is built from prog'REL using the linker. Letting co_prog be the name of the above CO, and assuming neither prog'XQT nor prog'REL exists, the command:

```
Configure(co_prog,prog'XQT);
```

would result in the compilation of prog'TXT and the linking of prog'REL. Actually, Configure does not interpret the command lines itself, but invokes the ACLI for this purpose.

In a real project, the "prog" objects would be abstract objects. That is, iterated versions of these objects would exist. From the point of view of configuration management, versions are necessary to permit the reconstruction of previous instances of the program. In the present example, assume that the only extant versions are prog'XQT.1.1, prog'REL.1.1, and prog'TXT.1.1. The CO object must also have versions, so let the current version be co_prog.1.1. The above invocation of Configure has then created the versions of the "XQT" and "REL" objects, and has set their history attributes. There are four history attributes attached to each created version:

date-time -- set to the date and time when the version was created.

dep-list -- set to the versions that the version immediately depends upon. For instance, the dep-list attribute for prog'REL.1.1 is:

```
prog'TXT.1.1,Ada.m.n,ACLI.p.q,
    Configure.s.t,co_prog.1.1
```

Note that the versions of all tools that were used, along with the version of the CO that was used, must be included. The Ada Compiler was used explicitly, while the ACLI and Configure were used implicitly.

ref-count -- incremented for each version that immediately depends on this version. For instance, the ref-count of prog'REL.1.1 is set to 1 since prog'REL.1.1 depends on it.

cfg-list -- set to include the name of each CO that references this version. This is an optional attribute that the user may request when an abstract object is created. For simplicity, the current example will not use cfg-list.

These history attributes provide a record of the steps that were performed in generating a given object version from the object versions upon which it immediately depends. The dep-list attribute enables the construction of a dependency graph. The complete derivation record of the object version that roots the graph is thus available from the history attributes of all versions in the graph.

Having available the derivation record of a version enables the information content of the version to be deleted and later reconstructed. Thus, for instance, in order to minimize the storage required for an infrequently used version, the object code portion of prog'XQT.1.1 may be deleted. The derivation record still would be preserved in the history attributes, enabling reconstruction via the call:

```
Configure(prog'XQT.1.1);
```

This minimization of storage becomes especially significant for large projects that must be maintained for long periods of time.

The present design results in relatively little information being stored in the history attributes, compared with designs that store the entire derivation record with each version. The key observation is that the derivation structure and commands in the CO tend to change far less frequently than the objects being configured. This stability is due to the fact that the CO represents the module structure of a program, and such structure is rarely changed over the program's lifetime. Thus the same version of the CO will provide derivation commands for a large number of configured object versions.

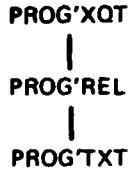
It must be noted that further economy is achieved by the storage method used for versions of text objects. Derivation histories are not maintained for

text objects (except when program generators are used). Instead, a version of a text object is indicated by a delta on the previous version. Delta storage eliminates much of the redundancy ordinarily inherent in versions of the same object. The algorithms for computing and storing deltas are described in X.Y.Z-KDBS.

Referring back to the example, suppose the programmer now edits prog'TXT to produce a prog'TXT.1.2. The next step is to bring the configuration up to date. This updating is accomplished by the command:

```
Configure(co_prog,prog'XQT);
```

Note that this is precisely the same command that was used to create the original version of prog'XQT. Configure first builds the dependency graph that is implicit in the dependency rules in co_prog. The graph is fairly simple in this case:



Examination of the graph yields the fact that the current version of prog'REL is older than the current version of prog'TXT. Thus a new version of prog'REL must be built, and the ACLI is invoked to execute the command line defining the compilation step. Next, it is determined that the current version of prog'XQT is older than the current version of prog'REL, so the ACLI is invoked again to execute the linking step. Thus each object in the configuration is brought up to date relative tc one other. Note that if a configuration is up to date, the above Configure call will examine the graph and report that nothing needs to be done.

3.3.2 A More Detailed Example

The dependency rules in a CO can define an arbitrarily complex directed acyclic graph. These rules are therefore sufficient for describing the compilation order that must be imposed on the modules of an Ada program. The following CO might be provided to specify the construction of the program given as Example 3 in Chapter 10 of the Reference Manual for the Ada Programming Language. (This program is reprinted in Figure 3-1).

```
procedure TOP is
     type REAL is digits 10;
     R, S : REAL := 1.0;
     package D is
          PI : constant := 3.14159_26536;
          function F (X : REAL) return REAL;
          procedure G (Y, Z : REAL);
     end D;
     package body D is separate;                    --  stub of D
     procedure Q (U : in out REAL) is separate;     -- stub of Q
begin    --  TOP
     Q(R);
     ...
     D.G(R,S);
end TOP;
```

```
separate (TOP)
procedure Q (U : in out REAL) is
     use D;
begin
     U := F(U);
     ...
end Q;
```

```
separate (TOP)
package body D is
     --  some local declarations followed by
     function F (X : REAL) return REAL is
     begin
          --  sequence of statements of F
     end F;
     procedure  G (Y, Z : REAL) is separate;       --  stub of G
end D;
```

```
with INPUT_OUTPUT;
separate (TOP.D)
procedure G (Y, Z : REAL) is
     --  use of INPUT_OUTPUT
begin
     --  sequence of statements of G

end;
```

Figure 3-1. Example of Constructed Program

```
— Configuration  co_top

top'XQT    : q'REL d'REL g'REL io_package'REL
     link((q,d,g,io_package,top),NAME=¬top);
q'REL      : top'REL q'TXT
     Ada(q);
d'REL      : top'REL d'TXT
     Ada(d);
g'REL      : d'REL g'TXT io_package'REL
     Ada(g);
top'REL    : top'TXT
     Ada(top);
```

The derivation graph implicit in co_top is illustrated in Figure 3-2.

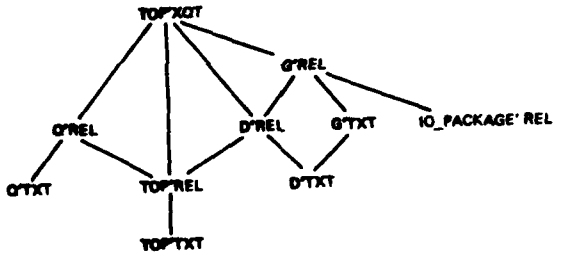


Figure 3-2. Derivation Graph

Given this graph, which Configure builds for its internal use, we can see the effects of changes to the constituent objects of the configuration. Assume that all objects are currently up-to-date. A programmer then edits and produces a new version of q'TXT, and calls Configure:

```
Configure(co_top,top'XQT);
```

The only recompilation required is that of q'TXT, followed by a relinking to generate an updated version of top'XQT. Configure will determine and

perform this minimal set of operations. Similarly, if a new version of d'TXT is created, Configure will call for a recompilation of d'TXT, a recompilation of q'REL, and a relinking to update top'XQT. Once the CO has been defined, the programmer is relieved of remembering which modules have been changed and which command lines need to be executed to process the changed modules.

Not all objects mentioned in a CO need to be defined, by a dependency line in that CO. The example refers to the object io_package'REL but does not mention what it depends on. Such objects will ordinarily be given the same treatment as text objects, and the date-time of their current versions will be examined. It is possible, however, to invoke Configure recursively to check the currency of these objects. If io_package'REL were defined in another CO, co_io, then the following dependency rule could be added to co_top:

```
io_package'REL      :
        Configure(co_io,io_package'REL);
```

Because in co_top the object io_package'REL depends on nothing, the command associated with this dependency rule will always be executed. The recursive invocation of Configure will, if necessary, update io_package'REL.

The current example exactly represents the compilation order required by Ada. For such cases, it is possible to provide further automation by adding a system tool that processes a set of Ada text objects and generates a CO. Often, however, the CO will contain nonstandard derivation steps. Such steps may specify test scripts and installation commands. For example, the following dependency rules may be added to the above CO:

```
test      : top'XQT sample'DAT
     test'DAT -¬ top'XQT();
install   : top'XQT
     copy(top'XQT,project'PTN);
```

The command line associated with the "test" dependency will execute the

current version of top'XQT with input sample'dat. Note that test is not an object, and will therefore never be considered up to date. Thus the call:

```
Configure(co_top,test);
```

will always run the test, whether or not a new version of top'XQT needs to be rebuilt. Similarly, install is not an object, so the call:

```
Configure(co_top,install);
```

will always result in the current version (or, if necessary, the rebuilt version) of top'XQT being copied to the partition project'PTN. This extension of the CO to include nonobjects allows Configure to be used to specify many additional operations that naturally occur in project development and maintenance.

INTERIM TECHNICAL REPORT

TAB 5

ADA COMPILER

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the Compiler preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements upon which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The Ada Compiler is one of the largest and the most complex of the MAPSE CPCIs. However, because its functions are dictated largely by the formal definition of the Ada Language, it is also the most formally defined system element. Its primary function, of course, is to accept an Ada source program and translate this program into a relocatable object program for a particular target computer. The target computer may be the host upon which the compiler runs, or a separate, possibly dissimilar, machine.

1.2 DESIGN PRINCIPLES

The AIE design has provided an opportunity to develop a sorely needed, but clearly achievable, language-oriented development system. One of the major goals established at the beginning of this effort was to take full advantage of this opportunity by developing an integrated set of MAPSE tools.

The precepts that guided the design of the Compiler are outlined below:

1. The full Ada Language must be implemented.
2. The Compiler must be designed to produce very efficient code for the required hosts and allow for very efficient code generation for future targets.
3. The Compiler must meld easily and cleanly with other MAPSE tools. All tools should use a consistent user interface.
4. The Compiler should promote total life-cycle cost reduction. It must be reliable and easily maintained. It must be retargetable without compromising code efficiency and rehostable without

sacrificing performance. Its development should allow for maximum reusable code across other tools as well as for other hosts and targets.

5. The Compiler shall be comprehensive. When a function may be performed as an inexpensive by-product of compilation and thereby eliminate the need for a separate tool and, hence, a separate process for the user, the function should be incorporated.

6. The Compiler shall be practical in its use of resources. It shall be sharable. Its minimum memory requirements should permit rehosting to a wide range of hosts. Its resource utilization should be parameterizable to maximize performance and capacity when increased resources are available.

7. The Compiler shall promote productivity. It shall be thorough in its error detection and explicit in its diagnostic reporting. It shall detect and alert the user, as early as possible, of potential compilation order violations.

8. The Compiler shall use an intermediate language approach that provides a clean interface for retargeting the Compiler. Additionally, this interface should be usable by other APSE tools.

SECTION 2 - BACKGROUND

The following section provides a description of the background of this design effort. Previous work and literature have contributed to the design of the Compiler.

2.1 PREVIOUS WORK

During the design phase, an analysis was performed on past compiler techniques and, to the extent possible, on the current Ada approaches. Although Ada provides a number of new language features, there is nothing that invalidates accepted and successful techniques for developing rehostable and retargetable compilers. The problems of generics, exception handling, overloading, and separate compilation will entail extensions to these techniques, but the design represents a low-risk, cost-effective approach.

Past efforts that have contributed to the proposed design are:

1. For retargetability/rehostability - Several families of proven rehostable/retargetable compilers including those for ALGOL 60 hosted on the UNIVAC 490/494, IBM 7090/7094 and GE 635 (Honeywell 600/6000) and targeted to the three hosts plus the XDS 910 and the CDC 3600; GENESIS/JOVIAL J3 compilers hosted on the IBM 360, CDC 6600 (and Cyber Series) and the HIS 600/6000 and producing code for those hosts and the Rolm 1666; and JOVIAL J73 compilers operating on the DEC-10/20, the IBM 360/370, and the UNIVAC 1100 series with code generators for the hosts, the AN-AYK/15 (Westinghouse HBC), the Delco Magic 362F, the Brassboard Fault-Tolerant System Computer, the MTL-STD-1750 and -1750A architecture, the Collins CAPS-7, and the TI 9900 series. This latter family is also being targeted to the PDP 11/70, VAX 11/780 and the Zilog Z8002.

2. For optimization - Numerous CSC-developed FORTRAN compilers that have been recognized as industry standards for optimization. The GENESIS/JOCIT J3 family of compilers has two to four global optimization passes from which much of the proposed design has been drawn.

3. For user-friendliness and comprehensive function - The CSC/SEA developed J73 compilers provide statistics collection, compool processing, source reformatting, debugging interfaces, relocatable output, source copy, compool procedure definition/reference compatibility, high payoff optimization, alphabetically sorted cross-reference/attribute listing, global system name concordance, and symbolic trace and walkback support.

4. For system integration - The manner in which the compiler fits into the overall MAPSE system and interfaces with other tools is modeled after the CSTS GPS environment.

2.2 RELEVANT DOCUMENTS

1. Goos, G. and G. Winterstein, Towards a Compiler Front-End for Ada, SIGPLAN - Symposium on the Ada Programming Language, December 1980.

2. Rosenberg, J., et al., The Charrette Ada Compiler, SIGPLAN - Symposium on the Ada Programming Language, December 1980.

3. Hisgen, A., et al., A Runtime Representation for the Ada Variables and Types, SIGPLAN - Symposium on the Ada Programming Language, December 1980.

4. Sherman, M., et al., An Ada Code Generator for VAX 11/780 with Unix, SIGPLAN - Symposium on the Ada Programming Language, December 1980.

5. Cornack, G. V., An Algorithm for the Selection of Overloaded Functions in Ada, SIGPLAN Notices, Vol. 16, No. 2, February 1981.

6. Ichbiah, J., et al., Rationale for the Design of the Ada Programming Language, SIGPLAN Notices, Vol. 14, No. 6, June 1979.

7. Hecht, M. S., Flow Analysis of Computer Programs, North-Holland, 1977.

8. Aho, A. V., and J. D. Ullman, The Theory of Parsing, Translations, and Compiling Volumes I and II, Prentice-Hall, 1972.

9. JOCIT/J3 Project Workbook.

10. JOVIAL J73 Maintenance Manual.

11. Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.

12. Tarjan, R. E., Depth-First Search and Linear Graph Algorithms, SIAM Journal of Computing, 1:2, 1972.

13. Loveman, D. B. and Faneuf, Program Optimization - Theory and Practice.

SECTION 3 - FUNCTIONAL DESCRIPTION

The following section provides a general overview of the Compiler interfaces within the MAPSE and describes the design tradeoffs performed during the Compiler preliminary design.

3.1 INTRODUCTION

This section discusses the system interfaces and functional design capabilities for the Compiler in terms of the design priciples outlined in Paragraph 1.2.

3.2 SYSTEM INTERFACES

The Ada Compiler interfaces with other programs in the MAPSE as described below:

ACLI - Although the Compiler may be invoked by any tool, the MAPSE user will normally perform compilations by an ACLI request that will establish a Compiler process.

KDBS - The KDBS is called by the Compiler through use of Ada Standard I/O to access source and library objects, to read and write temporary work files, and to produce the relocatable object and output listings.

MAPSE Tools - The Compiler will have indirect interfaces with other MAPSE tools through shared objects: Editor-Compiler for Ada source and listing objects; Compiler-Linker for program libraries and relocatable objects; and Compiler-Linker-Debugger for the Ada debug tables.

MAPSE Development System - Because the Compiler will be written in Ada and will be at least partially developed and certainly maintained under MAPSE, the compiler must conform to all design and development conventions necessary for program implementation and installation.

3.3 FUNCTIONAL CAPABILITIES

The Ada Compiler is the single MAPSE tool that must be used to obtain an executable program. Its function is to read Ada source from a text object or from the standard input file, include other text objects as directed by

pragmas, access information describing separate compilation units from Program Libraries, and produce a relocatable object.

3.3.1 Compiler Functions

As the program directly in the mainstream of software development, the Compiler is in an excellent position to perform many other useful development support and monitoring functions. The functions to be performed by the Compiler are (including usual compiler functions):

1. Full Ada Language translation
2. Comprehensive listing generation
    a. Original source listings with Editor line numbers
    b. Reformatted source listings
    c. Lucid diagnostic messages
    d. Statistics summary
    e. Cross-reference/attribute listing
    f. Side-by-side assembly listing
    g. Compilation summary
    h. Maintenance dumps and traces
3. Creation, use and updating of Ada Program Libraries
4. Generation of Symbolic Abstract Program Representation and Program flow description
5. Thorough, machine-dependent and independent optimization
    a. Common sub-expression elimination
    b. Loop Optimizations
        (1) Strength Reduction
        (2) Code redistribution
        (3) Loop collapsing
        (4) Loop control
    c. Constant computations and conversions
    d. Value folding
    e. Register name/value dedication

f. Dead code and variable elimination

g. Inline substitution

h. Exception checking

i. Discriminant validation

j. Space reclamation and management

k. Expression reordering

l. Subscript linearization

m. Boolean/relational optimization

n. Path merging

o. Instruction exploiting

p. Efficient calls and parameter passing

q. Register allocation

r. Peephole optimizations

6. Full target machine code acceptance

7. Debug table generation

8. Compilation order validation

9. Support for environment simulation, performance tuning and path entrance verification

10. Support for system wide compilation unit name concordance.

In addition to the functional requirements satisfied by the list above, several other requirements must be satisfied by the Ada Compiler. Those requirements include:

11. Retargetability
12. Rehostability
13. A compilation rate of at least 1000 statements per minute
14. Small programs compilable in 256KB
15. Reliable and maintainable.

3.3.2 Compiler Organization

The compiler is organized as a tree with a root containing compiler-resident subprograms. The remaining components of the Compiler are structured as logically related groupings with sequentially executing groups arranged as parallel or nested branches. The groupings (also known as phases) and a diagram illustrating the component structure are presented in Figure 3-1.

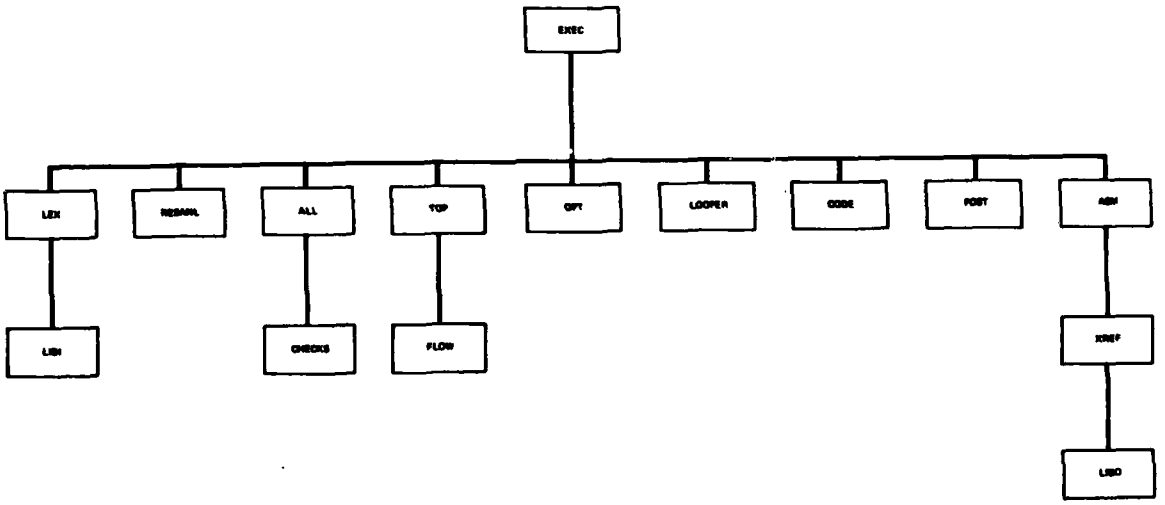


Figure 3-1. Ada Compiler Structure

3.3.3 Compiler Phases

The EXEC is the root phase of the compiler and contains the KAPSE interfaces and miscellaneous utility routines used throughout the compilation process. The functions performed by the EXEC phase are:

1. Compiler option setting and monitoring
2. Phase sequencing, loading and executing
3. Dump and formatting utilities
4. Space management packages
5. Symbol table routines
6. Diagnostic message handling
7. Listing header and pagination control

The remaining fourteen compiler phases, along with their respective functions, are presented below.

8. LEX

    a. Lexical analysis
    b. Syntax checking
    c. Name numbering
    d. Partial declaration and scope processing
    e. "With" and "use" name scope qualification
    f. Initial name resolution

9. LIBI

    g. Library unit name/attribute extraction

10. RESANL

    h. Declaration completion
    i. Final name resolution
    j. Semantic analysis
    k. Expression processing
    l. Statement processing
    m. Intermediate language generation
    n. Generic instantiation
    o. Generic and inline subprogram preservation

p. Separate body stub generation and symbol table checkpointing

q. Cross-reference file generation

11. ALL

r. Determination of object size requirements

s. Organization of packed variables

t. Allocation of variables

12. CHECKS

u. Constraint analysis

v. Flow label reference recording

w. Inline insertion

13. TOP

x. Target machine intermediate language transformations

14. FLOW

y. Subprogram/label reference count computation

z. Code straightening

aa. Data usage list generation

15. OPT

ab. Common subexpression elimination

ac. Value folding

ad. Constant arithmetic and conversion evaluation

ae. Code deletion

af. Inline procedure/function optimization

ag. Branch optimization

ah. Redundant constraint elimination

ai. Miscellaneous local optimizations

16. LOOPER

aj. Code movement

ak. Strength reduction

al. Test replacement

am. Loop collapse

17. CODE

an. Intermediate language-to-code transformations

ao. Logical register assignment

ap. Temp allocation

aq. Operand accessibility analysis

ar. Synonym creation and spoiling

18. POST

as. Code macro expansion

at. Register allocation

au. Code improvements

av. Path merging

aw. Conditional code resolution

ax Constant pooling

ay. Branch optimizations

az. Relative address optimizations

19. ASM

ba. Preset processing

bb. Relocatable object generation

bc. Formal assembly listing

bd. Final address resolution and control section allocation

20. XREF

be. Cross-reference listing generation

bf. Concordance name usage support

21. LIBO

bg. Program Library updating

bh. Debug table generation

The determination of phase versus memory load organization will be an implementation decision based upon resources and relative phase sizes.

3.3.4 Design Rationale

The following paragraphs illustrate how the design principles presented in Paragraph 1.2 have been achieved and justify the various design decisions made.

3.3.4.1 Full Language Implementation

The approach throughout this effort has been to design a compiler that accommodates the full Ada Language.

Recognizing that the Ada language is a new development and is pioneering new high-level language facilities such as generic subprograms, comprehensive exception handling, built-in tasking and nontrival operator and procedure overloading, a concerted effort is made to provide flexible language analysis algorithms that permit changes to the language in the advent that ambiguities or omissions are discovered.

3.3.4.2 Efficient Code Generation

Six compiler phases are dedicated to the production of efficient object code. Three of these phases are oriented to the specific target computer. A common problem with past retargetable optimizers has been that the global optimizations performed have not been tailored to the selected target computer. The proposed design achieves this by the inclusion of an optimizer prepass that transforms the intermediate language as necessary to orient the subsequent global optimizations toward the target computer. Additionally, static information describing the target resources will be used by the optimizer.

The machine-independent phases of the optimizer perform full program flow analysis, and, using flow-related variable set/use lists, perform the optimizations listed in Paragraph 3.3.1.

Perhaps the most important optimization that can be performed on a register computer is the global allocation and dedication of values and addresses to registers. This function has been accomodated by logically assigning registers during code generation and performing the dedication and allocation of registers in a subsequent phase using frequency counts (weighted by loop depth) collected by the optimizer and code generator.

Rather than produce simplified intermediate language out of the front end to reduce code generator retargeting costs at the expense of code efficiency, the Ada semantic operators will be represented in the intermediate language and TOP will tailor the intermediate language and simplify it for the particular target code generator. This allows such operations as slicing, loop control, boolean array logical arithmetic, membership and constraint tests, etc. using machine instructions where possible on the target, and simplifying such operators when not supported by the target.

No attempt is made to table-drive the code generator to enhance retargetability because of the decision not to compromise target code efficiency. Past experience indicates that table-driven and interpretive code generation approaches cannot match the traditional proven techniques in either compiler or code efficiency. Those that attempt to accommodate the full characteristics of the target instruction architecture and addressing facilities have tables that require equivalent effort and time to retarget. The drivers for such code generators must make the full spectrum of tests regarding the availability of features on the particular target - most of which will be false for any particular target, again resulting in poor performance. Finally, if history is to be our guide, the first new target is apt to have a characteristic that was not, and could not be, anticipated, thereby obsoleting, or at least compromising, both the driver and possibly the table formats.

3.3.4.3 MAPSE Tool Integration

The source text objects produced by the Editor are fully compatible with the Compiler. The Compiler will utilize in its listings any source line numbers or keys created using the Editor. Source correlation will be achieved by using these Editor line keys in the following listings: source, diagnostics, name cross-reference, machine-level side-by-side relocatable, etc. These same line keys will be passed to the Debugger in the Ada debug tables to again permit a stable cross-referencing of source program location.

Additionally, the compiler listings will be produced in the same text object format to allow for common text and listing routines and for convenient perusal by the Editor. To further support listing perusal, the compiler will produce a listing code associated with each listing line, which identifies the line by listing type: source, diagnostic message, cross-reference. This code will permit a user to easily list only certain line types such as diagnostic messages.

The Ada debug tables (ADTs) will be structured for efficient processing by the Debugger. Addresses contained in the ADTs will be marked to permit their relocation by the Loader. The Compiler will compute the legal implant addresses for the Debugger to minimize its target dependence.

A consideration in the design of program library and the ADTs produced by the Compiler is the anticipated requirement to provide a description of a program's data base to permit the development of environment simulation programs and data recording/reduction tools.

The Compiler provides information to the Linker to detect attempts to combine object programs with incompatible interface specifications. Such incompatibilities can arise because different descriptions were used or, for example, a called subprogram's specification was changed since a caller was last compiled.

Rather than actually perform stub generation at the occurrence of "is separate", the Compiler will create a null library entry indicating a body stub object. The Linker will replace all calls on body stubs by a call on a general target dependent body that will optionally report its entry, and raise an exception if the call had "out" parameters or was to a function subprogram.

The program library concept has been expanded to enhance Linker performance. By including all compilation units' preambles in the Library, the Linker may perform complete link allocation and name resolution without having to open and read each relocatable object. This library format minimizes double opening of the object files and reduces the number of files open simultaneously.

The program library is also used as the repository of other data created by the Compiler to support the global program concordance of compilation unit name references. This feature allows composite data to be collected by the Linker during partial linking and finally, a listing to be produced by which managers and programmers may quickly determine any program's use of independent subprograms, packages, and individual visible names within packages.

The Compiler will obey MAPSE conventions for diagnostic message formatting and control. Message brevity will be controlled, as with all tools by ACL commands. Listing options for the Compiler and Linker will be identical for similar functions.

An important function required during the development of any system is history maintenance. To support this function, the Compiler includes the generated relocatable object tracing information that describes the derivation of the object program. Included in this information are the versions of all input objects (original source, included source, packages and the Compiler itself).

The Compiler also conforms to MAPSE conventions for default naming and versioning of output objects.

3.3.4.4 Life-Cycle Cost Reduction

The Compiler will be written entirely in Ada. Strong typing and abundant use of types will be exploited to enhance reliability. Enumerated types will be used to maximize the readability of Ada programs.

To promote retargetability, a standard relocatable object module format is to be developed. This format is described as an Ada package whose definition is adapted for each target machine and therefore permits definition of any size target computer word. In addition, to minimize cost, a standard object module formatter is to be designed for the Compiler and Linker and delivered as an additional tool for use by later MAPSE programs that need to construct object modules. Similarly, a standard assembly lister capability reduces the cost of future retargetings.

The optimized intermediate language will contain redundant synonyms for values to permit simpler code generation by not requiring value/name tables as in several earlier optimizing compilers. TOP will transform the intermediate language from the front-end phases, RESANL and CHECKS, into an intermediate language tailored for the target and eliminate target-inappropriate operators.

Literal values will be maintained in a canonical form to allow compile-time computation and conversion routines to be rehostable.

Formatting and conversion routines, listing header and pagination packages, a standard diagnostic message formatter, and common command parameter scanners required by the Compiler will be developed as general purpose utilities to serve the needs of other MAPSE tools.

3.3.4.5 Comprehensive Functions

The Compiler will perform functions that eliminate the need for other tools. There are several utility functions available in language-oriented systems that have been incorporated as an integral part of the Compiler, thus eliminating the need for independent programs performing these functions. These are described in the following paragraphs.

3.3.4.5.1 Pretty Printing

To eliminate the need for a separate pretty printer, the Compiler produces, as an option, a reformatted representation of the input program. The structure of the program as indicated by declarations, program flow statements, subprograms, and blocks is made clearly visible by source indentation. Nesting levels are indicated for easy nest/unnest association. Source lines included by the INCLUDE pragma are displayed in accordance with the user option.

3.3.4.5.2 Statistics Collection

The Compiler will be instrumented to collect statistics during the compilation process. These statistics will be used to provide language feature usage frequency for optimizer tuning; error frequency for future language analysis, documentation improvements, and training emphasis; and correlation of language feature usage versus program checkout time.

To assist users in program checkout and performance tuning, the Compiler supports the Debugger facility for collecting dynamic statistics and timing information. Included in this facility is path entrance verification.

3.3.4.5.3 Program Flow Description

As a by-product of the extensive flow analysis performed by the Compiler for the purpose of global optimization, the Compiler will optionally produce a list of all basic blocks (regions of code entered only at the beginning and exited only at the end) and the paths to and from each block. Each block will also point to the beginning and end of the external intermediate program representation (assumed to be Diana as of the date of this document) associated with this block. This information is produced to accomodate path entrance statistics as well as automatic flow documenting tools.

3.3.4.5.4 Global Name Concordance

As an integral function of the set/use processor, XREF, and the library updater, LIBO, the Compiler will produce the data for a global compilation unit visible name concordance.

3.3.4.6 Practical Performance

The Compiler design has been selected to fit within modest host requirements. Design decisions that foster reduction of space requirements include: putting all literals in the intermediate language rather than the symbol table, spilling the name table after LIBI, flushing inactive symbol table scopes, representing expressions in an encoded form during optimizer expression analysis rather than the interphase intermediate language form, purging unneeded package symbol table entries, structuring the compiler in compact functional phases to reduce instantaneous memory requirements, duplicating common and redistributed expressions and synonyms in the intermediate language to eliminate the need for memory tables in the code generator, generating intermediate language and code macros from the larger phases to be expanded by trailing smaller phases, using an indexed symbol table record array structure rather than access records that require full address size links rather than the smaller entry indexes.

To maximize compilation rate, the design avoids use of table driven scans and interpretive forms of intermediate languages and code generation.

The optimizer utilizes a nested region analyzer scheme that has been found to be considerably faster than those based upon the scheme described by Tarjan [12] and the P-graph algorithm described by Loveman and Faneuf [13]. The LNRA scheme documented in the JOCIT/J3 Project Workbook makes two simplifying assumptions that permit a simple forward optimization pass. The assumptions made - that forward branches enclose conditional code and backward branches enclose a loop - will usually be true with the application of structured program concepts and as encouraged by the Ada Language. However, to ensure the validity of the assumptions, the intermediate language will be straightened on an intermediate language scope basis allowing the more efficient algorithm to be used.

On computers with abundant memory, the Compiler will be organized to increase its performance and capacity. The symbol table will be arranged to increase its space as additional memory is provided. The phases will be combined to reduce phase load and I/O requirements.

3.3.4.7 Productivity

The Compiler syntax and semantic analysis performs thorough error detection. In all but a few table overflow situations, error recovery will occur at a statement end or at a recognizable statement start. The diagnostic messages will be parameterized with user names and source symbols to explicitly describe the error detected. Pointers placed under the offending source line will assist the user in pinpointing the error. The message text will be carefully selected to aid the user in correcting the problem. Checks will be made to prevent errors from cascading throughout the program.

Statements found to have serious errors will be replaced in the Compiler by a "raise source_error" exception statement to permit partial testing of erroneous programs.

The Compiler will append information to the Program Library at each library unit use to permit the detection of compilation order violations and to support reporting of the required recompilations.

3.3.4.8 Intermediate Language (Diana) Usage

A major concern throughout this design has been the possibility of a forced intermediate language usage. The $TCOL_{Ada}$ and AIDA forms were studied initially and compared with the existing and proven intermediate language forms used in past compilers. The Ada semantic requirements were determined, as well as the requirements for thorough optimization, acceptable performance and retargetability. The potential needs of the other tools for an alternate representation of a program were considered. Late in the contract, another representation was developed - Diana. All representations seem to be oriented toward human consumption rather than compiler performance, code efficiency, or ease of retargeting. The design phase of this effort was, contractually, far too short to allow for the development of compiler algorithms to adjust to an evolving standard intermediate language. There is no question that a documented retargetable interface is required. However, it would seem that the intermediate language should be primarily designed for efficient processing by the Compiler.

The intermediate language to be used internally by the specified compiler will be oriented to the producing/consuming compiler phases. Operators and attributes that are present in one compiler phase's intermediate language may be totally absent in the next phase. The Retargetability Manual required as a deliverable of the implementation phase will document the particular compiler interface that will minimize a retargeting effort. This interface will likely be a memory-resident data base within the CODE phase that will be quite complex but will maximize the reusable code. This interface will not likely have a simple human-readable form but will be thoroughly documented and a cookbook approach to the effort required to add additional targets will be described. A capability will be provided by the compiler to transform, on option, its internal intermediate language into an external representation when a standard intermediate language form is settled upon. As of this writing this external representation will be Diana.

INTERIM TECHNICAL REPORT

TAB 6

LINKER

TABLE OF CONTENTS
Tab 6

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the Linker preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements on which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The MAPSE Linker is the tool used to combine the relocatable objects of several independently compiled program units into a single load object for loading and execution, or, with the partial link option, into a relocatable object for further linking.

1.2 DESIGN PRINCIPLES

The Linker is one of the basic tools of the MAPSE system and will be frequently used during the development and maintenance of the MAPSE itself as well as any program developed under the MAPSE.

The major design goals of the Linker are:

1. It shall support all Ada requirements.
2. It shall support the requirements of large program development efforts.
3. It shall be efficient and easy to use.

An efficient linker is an absolute necessity for large program development efforts. This requirement has guided the design of the Linker as well as the relocatable object and the program library format(s).

SECTION 2 - BACKGROUND

The following section provides a description of the background of this design effort. Previous work and literature have contributed to the design of the Linker.

2.1 PREVIOUS WORK

The basic design and functional capabilities of the Linker are derived from the development of the CSTS linker and experience with use of the linking facilities on several program development systems -- CSTS, CTS, TENEX, GCOS, TOPS-10, MULTICS, IBM-370 and others.

2.2 RELEVANT DOCUMENTS

1. Reference Manual for the ADA Programming Language, July 1980
2. Requirements for ADA Programming Support Environment, STONEMAN, United States Department of Defense, February, 1980
3. Statement of Work, Contract No. F30602-80-C-0292, 80 Mar 26.

SECTION 3 - FUNCTIONAL DESCRIPTION

The following section provides a general overview of the Linker interfaces within the MAPSE and describes the design tradeoffs performed during the Linker preliminary design.

3.1 INTRODUCTION

This section dicusses the system interfaces and functional design capabilities for the Linker in terms of the design principles outlined in Paragraph 1.2.

3.2 SYSTEM INTERFACES

Both the Ada compiler and the Linker will make use of a common relocatable object formatting package for formatting the respective output object.

Some targets may require that a special target-dependent load object format be produced. In these cases a program would be provided to transform the target-independent load object to the target specific format.

All programs developed under the MAPSE must be linked before they can be executed. Under user direction, either from the Link command or from a Link directive text object, the Linker reads program libraries and relocatable objects produced by the Ada compiler, or produced by the Linker itself from a prior partial link, and creates a single relocatable load object. The Linker will interface with the KDBS to access and update program libraries and relocatable objects.

3.3 FUNCTIONAL CAPABILITIES

The explicit Linker requirements specified in the SOW, STONEMAN and the Ada Reference Manual are minimal.

The common Linker functions are an implied requirement of the Ada language because of the need to create programs from separate, independently compiled compilation units. These Linker functions include the support of multiple location counters, the resolution of external references, the relocation of address references, the specification of symbolic equivalencies and name definitions and the evaluation of address equations.

In addition, the concept of a program library is defined in Ada to ensure that a program consisting of several independent compilation units will have the same degree of type safety as the same program submitted as a single compilation. The Compiler largely supports this compatibility requirement; however, the Linker further satisfies this requirement by validating the actual date of compilation of the objects in a link.

Further, the Ada language specifies a required order of elaboration of library units included in a program. The Linker satisfies this requirement by creating an elaboration procedure for each load segment; this procedure performs the required library unit elaboration in the proper order at execution.

Beyond these limited requirements the functions supplied by the MAPSE Linker are those that are required in a useful user-oriented program development system and that are available in some form in many commercial operating systems.

These functions are outlined below.

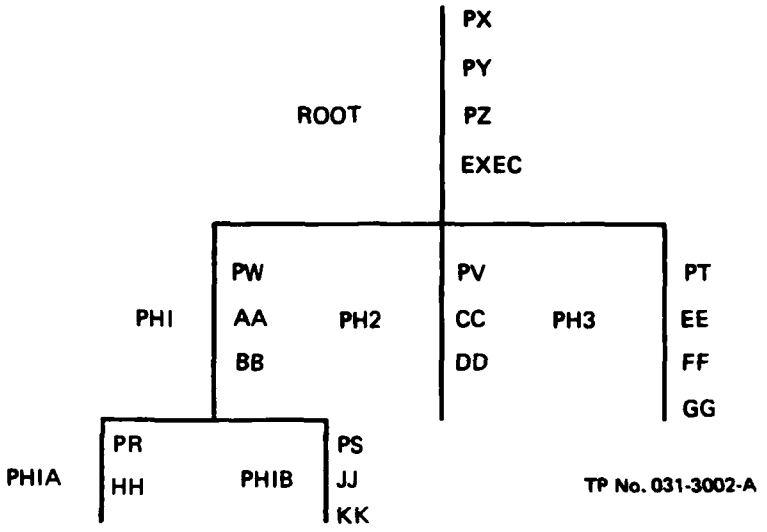
3.5.1 Partial Linking

To facilitate construction of large programs, it must be possible to link portions of the program independently, form them into partial link objects, and present them as input into a larger link activity. This idea is analogous to independent subprogram compilation. To support this, the Linker will construct linked objects in relocatable object format so that the Linker can accept its own output as input to a subsequent link. Furthermore, the user is able to specify in partial linking those external symbol definitions within the link that are to be retained as external symbols for resolution of references from outside the link. This approach results in a far more efficient use of the system resources by not requiring a total relink when only a few compilation units within an overlay segment are recompiled.

3.3.2 Multilevel Overlay Structure

Many linkers support only simple tree-structure overlays. This is clearly not sufficient for most large program organization efforts such as command/control systems or compilers. The MAPSE Linker supports this requirement by allowing the specification of a multilevel overlay program structure through simple linker directives.

As an example the following diagram pictures a simple two-level overlay structure. Names to the left of the overlay legs are used to represent segments, names to the right represent library units.



The link directives to build such a program are shown below:

```
ROOT    SEGMENT
        INCLUDE   PX, PY, PZ, EXEC
PH1     SEGMENT
        INCLUDE   PW, AA, BB
PH1A    SEGMENT
        INCLUDE   PR, HH
PH1B    SEGMENT   PH1A  --  overlay PH1A
        INCLUDE   PS, JJ, KK
PH2     SEGMENT   PH1  --  overlay on PH1
        INCLUDE   PV, CC, DD
PH3     SEGMENT   PH1  --  another overlay on PH1
        INCLUDE   PT, EE, FF, GG
        END
```

Following the execution of these link directives, the following library units will have been linked in the following order: PX, PY, PZ, EXEC, PW, AA, BB, PR, HH, PS, JJ, KK, PV, CC, DD, PT, EE, FF, GG.

3.3.3 Automatic Segment Fetching

In a program with an overlay structure it is necessary that the overlay segments be loaded before they can be referenced. This is supported by the MAPSE Linker as follows. Every call to a procedure located in a lower level overlay segment is replaced by an indirect call to a system routine that will check to see if the referenced segment is loaded. If the segment is not loaded, the referenced segment and any unloaded higher level segments will be loaded before executing the procedure call. Once the procedure (or the segment containing the procedure) has been loaded, the indirect reference will be changed to point directly to the subject procedure.

This approach allows the program overlay structure to be modified without requiring any source changes or any recompilations - only a new link is needed.

3.3.4 Heap and Stack Space

For each target system a suitable default size and location will be defined for the heap and stack space. Where the default is not adequate, the user may specify the size or the allocation of heap and stack space and whether or not the heap and stack space should share a common area or should be controlled separately.

3.3.5 Library Unit Elaboration

The Ada language has specific requirements concerning the order of elaboration of library units in the execution of a program.

The Linker must ensure that elaboration is performed for all library units in the program in the proper order. This is accomplished as follows.

Each compilation unit that requires specific elaboration has a callable elaboration prologue generated as a procedure by the compiler in a standard location in the code section of the object. For each object included in a load segment, the Linker finds its elaboration prologues (if any) and includes a call to it in an elaboration procedure for that segment at the start of the code section for the segment.

The elaboration procedure consists of a set of calls, in the proper order, to the elaboration prologues in the segment. When a segment is loaded, its elaboration procedure can be executed to perform the elaboration of every library unit in the segment.

3.3.6 Stub Generation

The Ada compiler produces a program library unit specification for every "is separate" procedure that identifies the procedure as being a stub. When the Linker includes such a stub procedure, it is replaced with a dummy procedure that simply returns. Optionally, the dummy stub procedure will raise a SOURCE_ERROR exception.

3.3.7 Compilation Order Validation

The Linker will perform the final verification of the Ada requirements for proper compilation order. All violations will be reported to the user although they will not abort the link.

3.3.8 Module Promotion

Where an object is referenced in two or more overlay segments of a program, that object will be promoted to a higher level segment that is common to those overlay segments and no others. If such a dominating segment does not exist, the referenced object will not be promoted.

If an object is explicitly included in a segment with the INCLUDE directive, that object will not be promoted. This allows the user total control over the program structure without relying on the linker defaults for implicit library unit inclusion.

3.3.9 Automatic Retrieval/Library Hierarchies

The user will be able to define a library hierarchy to specify the order of library searching for automatic object retrieval. This automatic retrieval is felt to be an essential element of a MAPSE Linker design.

The retrieval is based on the external symbols and library units referenced within a compilation unit and on the definition of those symbols that are contained in the program specifications of the user's program library hierarchy. For example, a user's program library hierarchy might contain the user's library, the project library and the system library. Anything unresolved in the user library will be resolved in the project library, anything remaining unresolved will then be resolved in the system library.

3.3.10 Boundary Alignment Module Placement

The user will be able to specify boundary alignment for any externally relocatable element of the link such as an object or location counter. The boundary alignment could be expressed as an absolute address or as some function of the next available location, such as double-word alignment, next byte, or next page. The user will be able to place objects in the linked program in a particular order allow the Linker to choose the order.

3.3.11 Linker Listings

The Linker shall produce user-oriented map and concordance listings. The map will show the allocation and the attributes of the various program location counters and entry points for each object and segment in the linked program. The concordance listing will show the referencing library unit name, the referenced library unit name, and the referenced element name for every external reference from a compilation unit. The order in which these names are applied in the sort may be user-specified.

INTERIM TECHNICAL REPORT

TAB 7

EDITOR

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the Editor preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the AIE contract. The requirements upon which the preliminary design is based are given in the System Specification (Type A) and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The MAPSE Editor provides the facilities for the creation and modification of text objects in the KDB. The capabilities provided include line- and string-oriented find, insert, delete, copy and move commands, object read and write commands, and command macro facilities.

1.2 DESIGN PRINCIPLES

The minimum functional requirements affecting the design of the Editor were specified in Paragraph 4.1.6 of the SOW.

The general design goals of the Editor, beyond meeting the SOW requirements are as follows:

1. The Editor must be simple and easy to use.
2. The commands must be natural, easy to learn and remember, but powerful and flexible.
3. The Editor must be a sharable, fully portable tool.
4. Support must be provided for minimal word processing and documentation generation.
5. Comprehensive editing facilities must be provided for line-oriented and screen-oriented devices.

SECTION 2 - BACKGROUND

The following section provides a description of the background of this design effort. Previous work in this area as well as literature have contributed to the design of the MAPSE Editor.

2.1 PREVIOUS WORK

The design of the MAPSE Editor has been influenced by experience with several editors on a number of different systems. The influence has been both positive and negative, and we have attempted to selectively include the good features of existing editors and to exclude the bad features.

A summary of some of these influences is shown below.

1. SOS - This DEC editor has had a major influence on the design of the MAPSE Editor. SOS has simple commands with an intuitive syntax; however, many of its commands have many, hard-to-remember options.

2. TECO - Several Versions - TECO has a cryptic but easily learned syntax. The comprehensive capabilities of Editor command macros, conditional and repetetive commands, and multiple commands per line were adopted from TECO.

3. CSTS - This CSC Editor has an unnatural syntax but its features of allowing a list of separate ranges over which an operation would apply and of allowing a fractional record key were adopted.

4. UCSD Pascal - The very natural screen-oriented editing features, the functions of the Environment specification and automatic indenting were adopted from this editor.

5. WYLBUR - The method of performing intralinear editing on a half-duplex system was adopted from this IBM editor.

6. GCOS - The ability to specify a long, complex string with a simpler string range syntax was adopted from this Honeywell editor.

2.2 RELEVANT DOCUMENTS

1. SOS Reference Manual
2. TECO Reference Manual (in DECSYSTEM10 User's Manual)
3. WYLBUR Reference Manual
4. Edm (in Multics Programmer's Manual)
5. Wordstar User's Guide, MicroPro International Corp
6. UCSD Pascal System Reference Manual
7. CSTS CPS Reference, Vol. 1: General

SECTION 3 - FUNCTIONAL DESCRIPTION

The following section provides a general overview of the Editor interfaces within the MAPSE and describes the design tradeoffs performed during the Editor design.

3.1 INTRODUCTION

This section dicusses the system interfaces and functional design capabilities for this system element in terms of the design priciples outlined in Paragraph 1.2.

3.2 SYSTEM INTERFACES

The Editor is basically an interactive tool that will communicate with the user through the standard input and standard output devices for the creation and maintenance of text objects.

3.3 FUNCTIONAL CAPABILITIES

The editor will interface with the KDBS to create and access text objects and for version control, and with the ACLI to allow ACLI command execution. There will be an interface to the user, interactively or through a command object.

The functional capability provided by the MAPSE Editor is summarized below, followed by some examples of the common editing commands and a sample editing scenario.

3.3.1 Basic Editing Functions

The Editor shall support line- and character-oriented find, insert, delete, substitute, copy, and move commands, screen cursor positioning commands, text object read and write operations, and parameterized command macro execution.

3.3.2 Line Numbers

Line numbers, which may be fractional, shall be supported as the record keys of a text file. This is required for a number of very commonly used line-oriented editing commands available in the Editor.

For Ada program development and testing, line numbers offer an additional advantage because they are supported and used as static reference points throughout the Editor, the Ada Compiler and the Debugger.

3.3.3 Primitive Word Processing

Realizing program development systems require the maintenance of numerous English language texts (system specifications, program documentation, status reports etc.) some minimal word processing capabilities are required. The following functions shall be supported in the MAPSE Editor: indentation, tabbing and margin facilities, token recognition, line filling, and justification.

3.3.4 Screen-Oriented Editing

A number of very useful and natural editing features are possible when the terminal interface is a screen device instead of a hardcopy or line-oriented device. These features shall be supported in the MAPSE Editor with screen-oriented editing capabilities.

3.3.5 Ada and English Language Token Recognition

In maintaining Ada source program and English language text it is often convenient to contextually recognize a sequence of tokens rather than an exact sequence of characters. Thus the string "SQRT(" would be equivalent to the token "SQRT" followed by the token "(" regardless of the number of blanks between the two tokens. Similarly, it is useful to recognize "the" only as a word rather than as part of "then" or "other". The Editor shall supply this feature by allowing exact character string searching and contextual token searching.

3.3.6 Command Macros

Command macros that consist of a set of normal Editor commands with optional parameters that may be invoked during editing provide a very simple and easily used capability that has proven highly useful in many systems. Provisions shall exist in the Editor for executing such command macros.

3.3.7 Environment Setting

The Editor shall allow the default environment parameters of the editing process to be established and modified to tailor certain editing and terminal characteristics for a user. Some of these parameters are: screen size, special characters (line-terminator, escape-character, cursor-movement-character etc.), verify flag, line increment, token matching flag, case matching flag, and indentation flag. In normal use, it is expected that a standard set of environment settings will be initialized transparently when a user invokes the Editor. The individual environment parameters may be changed during an edit session.

3.3.8 Cut and Paste-File Operations

The Editor shall support the accessing of external files and Editor "work buffers" for the reading and writing of portions of text to facilitate cut-and-paste and program construction operations.

3.3.9 Sample Editing Commands

Some of the basic Editor commands are listed below with examples of their use. These examples represent the general flavor of the command language and show some of the options.

3.3.9.1 Copy

Copy transfers text from one place to another and retains the original text.

C200..260	Lines 200 thru 260 inclusive are copied into the current position
C50@211	Line 50 is copied into the current buffer after line 211
C75..99@!Q	Lines 75 thru 99 are copied to buffer Q.

3.3.9.2 Delete

D100	Line 100 is deleted
D650..680	Lines 650 thru 680 inclusively are deleted

3D	Three lines starting at the current line are deleted
D10,20..40,60	Line 10, lines 20-40 inclusively and line 60 are deleted
D840#3..7	Character positions 3 thru 7 from line 840 are deleted.

3.3.9.3 Find

FABC/	Scan forward for the next occurrence of the string ABC starting from the current position
3FXYZ/	Scan forward for the third occurrance of the string XYZ starting from the current position
FP↵R/100..300	Scan forward for the string PQR starting at line 100 and searching through line 300 inclusively
FP(%*);/	Find the string starting with P(and ending with); starting from the current position. Thus, for example, the string P(A+2,SIN(Y)I**J); would be found.

3.3.9.4 Token Search (Set in Environment)

FALPHA'SIZE/	In the token search mode, this will find the next occurrence of the three consecutive tokens - ALPHA, ', and SIZE - regardless of how many blanks (if any) separate them.

3.3.9.5 Insert

L200I	Insert the specified lines after line 200

xxxx {One or more lines}/

The Editor supplies line number prompts, xxxx, for each line to be inserted

FABC/ Find string ABC

IXYZ/ Insert string XYZ after the string ABC found by preceding command.

3.3.9.6 ACLI Command

ACLI commands can be executed with the "K" directive. For example,

KLIST(//RADAR);/ This would invoke the utility LIST to list the contents of the partition /RADAR. Note that two slashes are required to represent a single slash in a string.

3.3.9.7 Print

P100..200 Print lines 100 thru 200 inclusively

P325 Print line 325

P50$5 Print 5 lines starting at line 50

P Print the current line

5P Print 5 lines starting at the current line

P10,40,70..90 Print lines 10, 40 and 70 thru 90.

The syntax allows more complex text ranges to be printed. For example;

P%B3.3.1/..%B3.3.2/ Print the contents of section 3.3.1.

3.3.9.8 Move

Move transfers text from one place to another and deletes the original text.

M540 Move line 540 to the current position then delete line 540

M750..780@1300 Move lines 750 through 780 inclusive into the current buffer after line 1300 then delete 750 thru 780.

3.3.9.9 Numbering

N50..90,60,2	Number lines 50 through 90; start numbering at 60 in increments of 2
N498..520,500,5	Number lines 498 through 520 inclusively; start numbering at 500 in increments of 5
N500..>,5000,100	Number lines 500 through the end of file start numbering at 5000 in increments of 100.

3.3.9.10 Quit

Q	Quit and terminate the Edit session. If the file being edited contains modifications, the user will be requested to confirm the request since otherwise the modifications would be lost.

3.3.9.11 Read

RTESTA/@300	Insert the entire file TESTA into the file being edited after line 300
RTESTB/50..210	Insert lines 50 thru 210 inclusively from file TESTB into the current cursor position.

3.3.9.12 Substitute

SABC/DEFG/	Substitute the first occurrence of the String ABC with the string DEFG starting at the current position
5 SZYX/WV/	Substitute the first 5 occurrences of the string ZYX with WV starting at the current position
SUT/SR/200..300	Substitute every occurrence of UT with SR in the lines 200 thru 300 inclusively.

3.3.9.13 Write

W	Write the current buffer to the current output object
WSAVEA/	Write the current buffer to text object SAVEA
W!P500..800	Write the lines 500 through 800 inclusively from the current buffer to buffer P.

The typical commands to terminate an editing session will be

W	Update the current text object
Q	Exit from the Editor.

3.3.10 Sample Editing Sessions

Two sample editing sessions are shown below. The first creates a new file, the second updates the file. Assume automatic indenting and line increment of 10. Carriage returns are not shown.

3.3.10.1 Initial Creation Session

```
EDIT (RESO);
I
10   Task Body RESOURC is
20   BUSY:  BOLEAN=FALSE;
30   begin
40        loop
50        select
60             When not Busy=>
70             Accept SIEZE do
80                  BUSY=TRUE;
90             end;
100       or
110       accept RELEASE do
120            BUSY:=FALSE
```

```
130       end;
140            pt
150       ej
160                 e
170            when not BUSY=>terminate;
180       end loop;
190  end RESOURCE;/
W
Q
```

Note: The line number prompts were supplied by the Editor.

3.3.10.2 Text Update Session

User Commands are underlined. Assume a line increment of 1 and verify mode.

Command	Explanation
EDIT (RESO);	Invoke the Editor for text object RESO
P10	Print line 10
10 Task Body RESOURC is	Line 10 is displayed
SURC/URCE/	Fix Misspelling
10 Task Body RESOURCE is	Line 10 shows correction
S=/:=/	Fix assignment symbol
20 BUSY: BOLEAN:=FALSE;	First = was found on line 20 and substituted with :=
D150..160	Delete garbage lines
140 pt	In verify mode,
170 when not BUSY=>terminate;	lines bounding deleted lines are displayed.
Spt/or/140	Fix typographical error on line 140
140 or	Line with correction displayed
L170I	Insert new line after 170
171 end select;/	Inserted line

SBOL/BOOL/20	Fix misspelling on line 20
20 BUSY: BOOLEAN:=FALSE;	Correction displayed
L120I;/	Insert missing ; at end of line 120
120 BUSY:=FALSE;	
L0	Locate line 0 (beginning of file)
99 (F=/)	Find many =
20 BUSY: BOOLEAN:=FALSE;	Lines containing = are
60 When not BUSY=>	displayed
80 BUSY=TRUE;	
120 BUSY:=FALSE;	
170 When not BUSY=>terminate;	
S=/:=/80	Fix = symbol in error on line 80
80 BUSY:=TRUE;	
W	
Q	Save the edited file and quit

INTERIM TECHNICAL REPORT

TAB 8

DEBUGGER

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

This part of the Interim Technical Report presents an overview of the Debugger preliminary design, the basic design principles involved, and the rationale for the decisions made during Phase I of the Ada Integrated Environment contract. The requirements on which the preliminary design is based are given in the System Specification (Type A), and details of the preliminary design are given in the corresponding B5 Specification.

1.1 FUNCTIONAL SUMMARY

The MAPSE Debugger supports program developers by providing comprehensive symbolic interactions with an executing Ada program. Facilities supported by the Debugger are Ada data object examination and modification; controlled execution in the form of single stepping, breakpointing and/or interrupting; and monitoring and tracing of program execution. The Debugger performs its function by direct execution of the subject process controlled by instruction implants rather than by simulation.

1.2 DESIGN PRINCIPLES

The principles that influenced the design of the Debugger are:

1. The Debugger must allow the programmer to debug at the level the program was developed, namely at the Ada source level.
2. The Debugger must support the checkout of operational, fully optimized programs.
3. The Debugger must be efficient and not require excessive memory or processor resource usage.
4. The user need not anticipate the existence of program errors and their location in order to correct them effectively.
5. The Debugger must support the checkout of programs that utilize the full Ada language including Ada tasking.
6. The Debugger must support the checkout of programs compiled for other target computers and their checkout in both the host environment and through use of simulators in the host environment.

7. The Debugger must support the location and correction of program errors and the performance tuning and quality assurance aspects of program development.

SECTION 2 - BACKGROUND

The following section provides a description of the background of this design effort. Previous work and literature have contributed to the design of the Debugger.

2.1 PREVIOUS WORK

Two major considerations are involved in the design of a user tool such as a Debugger. First, of course, it must provide comprehensive facilities with a user-friendly interface that enhances productivity and minimizes errors. Second, its performance and resource utilizations must not be such as to discourage its use. Several past models were studied for these characteristics. These are:

1. CSTS Program Checkout Facility (PCF)
2. DEC-10 Dynamic Debugging Tool (DDT)
3. Software Design Verification System (SDVS)
4. JOVIAL Interactive Debugger (JID)
5. UNIX™ Source Debugger (SDB)

2.2 RELEVANT DOCUMENTS

1. CSTS GPS Reference, Vol. 1: General
2. DEC System-10 Assembly Language Handbook
3. Computer Program Development Specification for Jovial Interactive Debugger, Type B5.
4. UNIX User's Manual, Dolotta, T. A., S. B. Olsson, and A. G. Petruccilli, Release 5.0, Bell Telephone Laboratories, June 1980.

SECTION 3 - FUNCTIONAL DESCRIPTION

The following section provides a general overview of the Debugger interfaces within the MAPSE and describes the design tradeoffs performed during the Debugger preliminary design.

3.1 INTRODUCTION

The Debugger is a functional part of the ACLI and is invoked when a debugging command is encountered in an ACLI command stream. The Debugger operates as part of the ACLI process and permits a user to control and monitor a child process of the ACLI by interacting with the Debugger through use of the Debugger directives.

The Debugger permits referencing of program identifiers through the use of ADTs produced by the Compiler and included by the Linker in the load object.

3.2 SYSTEM INTERFACES

This paragraph identifies the Debugger interfaces and their purposes.

ACLI-The Debugger forms a part of this MAPSE Tool and is invoked as a result of the ACLI encountering a Debugger directive. The ACLI and the Debugger cooperate during the processing of directives to permit each to utilize the services of the other.

KDBS-The KDBS is called directly and through the standard I/O package to perform directive input from standard input and to output various Debugger responses to standard output and standard error listing files. Standard I/O is also used to access the ADTs from the load object file as required.

KFW-The KFW is called to load the ADTs from the load object of the process to be debugged; to bind the two processes together to ensure that they are available to each other during the debugging operations; to control the initiation, priority, and resumption of the debug process; to establish addressing access, memory mapping, and memory protection to the debug process; to acquire additional workspace; and to query and manipulate

Process Control Blocks in order to test, report on and change the execution context; register settings, program counter, stack pointers, etc.

Ada Compiler-The Compiler produces the ADTs used by the Debugger to relate the debug process code and data to the Ada source names and attributes. The code produced by the Compiler represents another interface that the Debugger must understand to perform its functions. These interfaces embody the conventions followed by the executing program for subprogram calls and parameter passing, for exception handling, for space management, for Ada tasking, for local and heap data referencing, for register usage, and for optimization.

Linker-The Debugger interfaces with the Linker through the format and content of load objects. The primary information used by the Debugger from the load object is contained in the ADTs. The Linker relocates any addresses occurring in the ADTs.

User-The Debugger presents a system interface to the user for Debugger functions by utilizing a Debugger directive language (DDL). Ada program expressions, statements and objects are referenced in the DDL using Ada compatible syntax. The DDL is oriented toward usage in an interactive environment but is accepted identically from interactive terminals, directive objects or batch devices.

3.3 FUNCTIONAL CAPABILITIES

The Debugger provides the user with the interactive facility to discover and correct Ada program errors. The functions supported by the Debugger are requested by directives entered through the standard input file (normally the user's terminal). The Debugger controls the execution of the subject process by entering instructions into the user's program as a result of user requests. This debug process and the Debugger operate synchronously -- with execution of the debug process proceeding until an implant causes the Debugger to take control. When the Debugger is in control, the user may examine or set program variables, make procedure calls, enable execution traces, etc., and then resume execution of the process.

3.3.1 Debugger Directives

The Debugger directives use a simple syntax with Ada symbols for Ada program names, expressions, and values. The formats of these directives are summarized below:

load object_name (parameter_list)

Used to prepare a program object for execution. The parameters are specified using the ACLI conventions.

goto [address] [, {s i} integer_expression]

Used to begin (or resume) execution. If the address is omitted, execution will begin at the program entry point (or resume where last interrupted). This directive may also be used to execute a specified number of statements or instructions.

use package_name {,package_name}

This directive makes a name declared in a package referenceable without the package qualifier.

expression ?

The directive computes and displays the value of the expression or variable. Values displayed will be formatted according to the expression type. When displaying a variable, applicable indices or component names will be displayed.

variable_name := expression @

This directive is used to set a variable to a value (of the expression) or to display its address (if the @ is present).

procedure_name [actual_parameter_part]

Used to call an Ada procedure. Upon completion, control returns to the Debugger for the next directive.

trace [,{p f s a i w o}] [address][,file]

Used to trace execution of the program or Standard I/O operations on a file. The letter options are p for subprograms, f for flow paths (sequences of statements

uninterrupted by flow control semantics), s for statements, a for assembly language level, i for I/O operations w for subprogram walkback, and o for turning off indicated traces. The address is the limiting address of the trace. The file may be used only with I/O option.

trap[,s] variable address_range
trap[,o] [number]

This directive is used to detect and report each time a value within the variable or address_range changes. The s option causes execution to stop when the value has changed. The o option is used to turn off a trap.

dump [,{i r b o h c a s}] address_range

Dumps the address range specified in the format requested: integer, real, Boolean, octal, hex, character, assembly instructions, hardware status.

search address_range, {expression range} [,mask]

Searches for a value equal to the expression or within the range. A mask may be applied before the comparison.

time[,p f r] subprogram_name {,subprogram_name}

Causes timing information to be collected for the subprogram (or at the flow path level). The r option produces a summary report of the collected information.

flow[,r] subprogram_name {,subprogram_name}

Counts path entrance frequency within the subprogram. The r option prints a summary of the counts by subprogram.

insert address; "directive {;directive}" ?

Effectively inserts the directives at the indicated program address. If the operand is ?, the active inserts are displayed.

cancel insert_number

Cancels a previous insert.

record[,r w] file, variable { , variable }

Used to write a variable to a file or to read a value into a variable. May be used to support environment simulation or post data reduction.

print object_name [line_range]

Used to list text objects such as Ada program source. By executing the Editor, the user may browse through his source program or enter his modifications as debugging proceeds.

stop [,{s p}] [expression]

Stops execution, displays execution status, active procedure call chain, and, on option, the value of expression.

quit

Terminates Debugger commands and returns to ACLI.

In addition, Ada raise, abort, and delay statements may be used as directives. In the above directives, Ada definitions are used. An address may be an Ada name, an Ada program statement or a machine address. Its syntax is:

address := access_variable ± expression
@ name ± expression
@ integer
$ line_number

3.3.2 Rationale

The following paragraphs discuss how the design principles were achieved.

3.3.2.1 Ada Source Level Debugging

The Debugger directives and the Ada name referencing are expressed in Ada syntax. All values whether expressed in a directive or displayed by the Debugger will be represented according to type using Ada literal syntax.

Although the intent of the Debugger is oriented toward checkout to the extent possible at the Ada Language level, circumstances are expected to arise where machine-level observation is required. The Debugger supports this by permitting access to individual instructions, actual addresses, single stepping, instruction tracing and dumps. Even at this level, the Debugger attempts to allow references and display values in as symbolic a manner as possible.

3.3.2.2 Debugging Optimized Operational Programs

Although the Compiler supports an option to suppress optimization, thus ensuring the user of strict correlation of the source program to the executing program, the Compiler and Debugger have been designed to permit the checkout and correction of operational, optimized programs. The Compiler produces detailed messages that describe the optimizations performed. The ADTs also include descriptions of the optimizations performed with each statement. The Debugger will then be able to alert the user when debugging requests may be affected by optimizations.

3.3.2.3 Resource Utilization

The ADTS that provide the Ada names and attributes are maintained in the load object and do not occupy any memory until an Ada name is referenced. Only small portions of the ADTs are required at a time, thus the memory impact of symbolic name referencing is minimized.

No hooks are required in the code to support debugging. This eliminates the problem of significant code expansion often associated with debugging.

To minimize performance impact on a program being debugged, direct execution is utilized rather than interpretation. Execution is controlled by implanting breakpoints into the program as required. Furthermore, the Debugger operates as a separate process to prevent the results of the program being debugged from being affected unexpectedly.

3.3.2.4 Error Anticipation

Since error locations cannot be anticipated, the Debugger has been oriented to debugging without hooks and without special preparation of the program

for execution. An operating program may be interrupted and debugged with the same facilities available as if it had been loaded and started by the Debugger.

3.3.2.5 Full Language Support

All language features are supported by the Debugger through directives; however, the full Ada expression and statement capability is not provided within the Debugger directives to eliminate the need for full interpretation and the associated debugging performance penalty. The user may interface with and monitor Ada tasks. All Ada types are supported and all Ada program objects may be referenced and examined.

3.3.2.6 Target Program Checkout

The Debugger is designed to permit checkout of programs compiled for target computers being simulated on the host development computer as well as programs being directly executed on the host. The Debugger interfaces are fully standardized; the fact that the program is executed through simulation is transparent to the Debugger.

The Debugger also provides a mechanism to perform data recording and environment simulation through the record directive. By the use of minimal scaffolding and this record function, both static checkout and post data reduction facilities are provided.

3.3.2.7 Timing and QA Considerations

An often neglected development requirement is an automated mechanism to support program validation and refinement. Two features have been included as an integral part of the Debugger to support this need. These features allow the user to time individual program regions and count region entrances thereby providing performance tuning data as well as ensuring that all program paths have been exercised. The data required to support this facility is a fallout of the compiler optimizer.

3.3.3 Debugging Session

The following Ada program and debugging session illustrates the use of the directives.

```
 1  package trig is
 2  type deg is integer range 0 .. 360;
 3  type sc is float range -1.0.. 1.0;
 4  function sin(d:in deg) return sc;
 5  pi:constant := 3.14159;
 6  end
 7
 8  package body trig is
 9
10  function sin(d:in deg) return sc is
11      i:integer range 1..20;
12      limit:integer := 2;
13      factor:float := pi* float (2)/180.0;
14      term:float := 1.0;
15      sin:float := 0.0;
16  begin --use Taylor series to compute sine
17  series:  while i in 1 .. limit loop
18           term := term * factor /float (i);
19           if i mod 2 = 0 then
20              term := -term;
21           else sin := sin+term;
22           end if;
23      return sin;
25  end sin;
26  end trig;
```

The directives below represent a sample debugging session (the comments on the right are editorial notes). Debugger responses are underlined.

```
load trig                              --loads package trig
go                                     --elaborates package
use trig                               --default name qualifier
sin (30) ?                             --compute expression
= 0.52359                              --not very precise
insert $19,"i ?"                       --sin.i when encountered
insert #1                              --1st insert
sin (30) ?                             --try again
trig.sin.i=1                           --1st loop iteration
trig.sin.i=2                           --2nd loop iteration
= 0.52359                              --still wrong of course
insert @ sin.series;"limit:=20"        --series should have 20 terms
insert #2                              --2nd insert
cancel 1                               --disable 1st insert
sin (30) ?                             --one more time
=0.50001                               --much better
quit                                   --done, go correct source
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DA
FILM